Computer Networks 177 (2020) 107326

Contents lists available at ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet

Assisting reachability verification of network configurations updates with NUV

Yahui Li [a], Zhiliang Wang [b,c,*], Xia Yin [a,c], Xingang Shi [b,c], Jianping Wu [a,c], Fangdan Ye [a], Jiangyuan Yao [d], Han Zhang [e]

[a] Department of Computer Science and Technology, Tsinghua University, China
[b] Institute for Network Sciences and Cyberspace, Tsinghua University, China
[c] Beijing National Research Center for Information Science and Technology (BNRist), China
[d] College of Information Science and Technology, Hainan University, China
[e] School of Cyber Science and Technology, Beihang University, China

ARTICLE INFO

Keywords:
Network configuration update
Reachability verification
Formal methods

ABSTRACT

Configuring a network is always difficult and error-prone because of low-level configuration languages and complex routing mechanisms. Most network outages occur when network the configuration is updated. Thus, it is important to proactively verify network configurations. Unfortunately, there are two practical obstacles that inhibit the performance of existing configuration verification tools: (i) Lack of knowledge. Network users often do not know which input verification queries need to be verified. (ii) Limited scalability. Even the state-of-the-art tool (i.e., Minesweeper [1]) takes approximately 500 seconds to complete a single reachability query for a network with tens of routers. Considering the total number of queries that must be verified, real networks often make such techniques impractical. In this paper, we propose NUV, a framework for performing network configuration update verification. NUV outputs verification queries for only the endpoints whose forwarding behavior has changed under the updated network configuration. Based on the network forwarding model, NUV infers the potential traffic impact and eliminates endpoints whose forwarding behavior is equivalent in both the original network configuration and the updated configuration. The NUV outputs can be used as input to a rich collection of existing configuration verification tools. Evaluations on a series of benchmark networks show that NUV can solve the lack of knowledge problem, and it enables query verification to execute at speeds orders of magnitude faster than existing tools; thus, it also improves verification scalability.

1. Introduction

Computer networks often depend on a variety of distributed routing protocols to determine how traffic gets forwarded to its destination. The configurations on each router determine which protocol it can use, which neighbor should be sent route advertisements, which advertisements it should filter, and so on. To achieve user-customized policies (e.g., “node A either can or cannot talk to node B. ”), configurations must be designed and applied carefully. Unfortunately, due to the use of low-level configuration languages and the complex relationships among routing protocols and mechanisms, configuring networks is always difficult; consequently, configuration errors inevitably occur, some of which can cause the network to violate user policies [2]. Specifically, *most* network outages occur when the network configuration is *updated*. According to a survey of Alibaba’s network, configuration updates occur frequently due to incident mitigation, network changes, new commercial requirements or security concerns. In general, configuration updates may occur up to 2000 times per year, and more than 56% of network incidents are caused by configuration updates [3]. Thus, a new configuration should be verified before it is applied to the network.

An increasing number of applications that address configuration verification and control plane bug-finding (e.g., Batfish [4], ARC [5], ERA [6], and Minesweeper [1]) have been proposed. Through enhanced configuration parsing techniques and semantic encoding of the network control plane, these tools enable users to expose configuration errors through *reachability verification queries* (e.g., “can traffic from node A arrive at node B? ”), as well as a wide range of other common queries expressible in terms of reachability relationships, such as loop-free forwarding, bounded path length, or waypointing. However, there are two obstacles that prohibit these tools from working well. (i) *Lack of knowledge*. Reachability queries can be thought of intuitively as “who can reach who”. According to a survey of network administrators [7], network users often do not know which queries to use for verification; thus,

* Corresponding author.
E-mail address: wzl@cernet.edu.cn (Z. Wang).

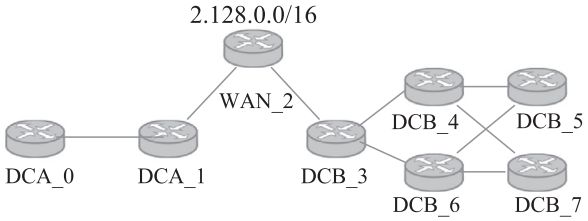
https://doi.org/10.1016/j.comnet.2020.107326
Received 28 October 2019; Received in revised form 23 April 2020; Accepted 18 May 2020
Available online 22 May 2020
1389-1286/© 2020 Elsevier B.V. All rights reserved.

a major impediment to these verification tools is determining which queries to execute. Because of this lack of knowledge, administrators may even perform *all-pair reachability queries* for the traffic from each possible endpoint to every other possible endpoint in the network [1]. From our survey of many network operators managing a global scale network of a large company or famous operator networks, even though some operator experience-based heuristics can be used to rule out some impacted traffic, the number of reachability queries to be verified can still be enormous. (ii) *Limited scalability*. To date, some network data plane verification tools, such as P-Rex [8], can perform fast verification for a single reachability query in polynomial time. However, these tools use router tables as input; thus, they cannot analyze reachability and detect errors before new network configuration files are applied. In contrast, network control plane verification tools analyze the network configuration files and can perform verification proactively. However, for a network with tens of routers, the state-of-the-art control plane verification tool (i.e., Minesweeper [1]) can require up to 500 seconds to perform a single reachability query [9]—and it is well known that the query verification time for the network control plane increases exponentially with the size of the network [9]. Consequently, verification tasks can easily become computationally expensive. Considering the total possible set of tasks (i.e., queries for near-all-pair queries), such techniques are impractical in many real networks.

To address the above problems, in this paper, we propose a framework named NUV based on the following observation. Configuration updates are always incremental rather than reconstructive. Consequently, only a few traffic events tend to be impacted because of configuration updates, rather than *all* the network traffic. Specifically, users typically want to make several changes to the existing configuration (e.g., updating access control lists (ACLs), adding static routes, updating route filters) rather than replacing the original configuration with a completely new configuration. By performing *queries* only on the *impacted traffic*, on the one hand, we are able to address the lack of knowledge question, and on the other hand, we can improve poor scalability by reducing the number of queries needed for the total verification task.

Consider the toy example in Fig. 1, where *DCA* and *DCB* have two internal routers (DCA_0 and DCA_1) and five routers (DCB_3 to DCB_7), respectively, which run the border gateway protocol (BGP). Both DCA_1 and DCB_3 connect to the external neighbor WAN_2 via BGP. WAN_2 has several services hosted inside the prefixes 2.128.0.0/16. We now update that configuration by adding a route map as2_to_as1 on router DCA_1 that applies the routes sent from its BGP neighbor WAN_2. The configuration in Fig. 1 is broken into blocks of statements, i.e., *configuration snippets*, and lines 13–18 are related to the route map as2_to_as1. The route map denies route announcements for any prefix that matches the first 16 bits of 2.128.0.0. Because announcements for (most) destinations do not interact, only traffic whose destination (i.e., WAN_2) matches the route map will be impacted. In such cases, the verification task can be reduced to just seven queries for traffic (from other nodes to WAN_2). In contrast, without knowledge of the impacted traffic, we would need to perform all-pair queries (each source to each destination) in the network.

Based on the above observations, we derive queries for only the traffic whose forwarding behavior has changed under the new configuration. To achieve this goal, we need to determine how configuration updates impact traffic. For each traffic, we can check whether its forwarding behavior in the new configuration is the same as that in the original configuration; however, this task is challenging due to the following factors. (i) Complexity. Traffic forwarding behaviors are determined by all router configurations, including a variety of protocol standards (e.g., BGP, OSPF, ISIS), complex protocol intersections (e.g., route redistribution), filters (e.g., route maps, ACLs) and so on. Different configurations impact traffic forwarding *in different ways*; consequently, the relationships between impacted traffic and updated configurations are extremely complex. (ii) Scalability. Recomputing the forwarding behavior for each traffic under a new configuration is intractable because it




```
----------// Configuration of DCA_1----------
1  hostname DCA_1
2  interface Loopback0
3    ip address 70.0.17.0/32
4  interface Serial0
5    ip address 10.0.31.1/31
6  interface Serial1
7    ip address 10.0.14.0/31
8  router bgp 1
9    neighbor 10.0.31.0 remote-as 1
10   neighbor 10.0.31.0 send-community
11   neighbor 10.0.14.0 remote-as 2
12   neighbor 10.0.14.0 send-community
13  address-family ipv4
14   neighbor 10.0.14.0 route-map as2_to_as1 in
15  exit-address-family
16 ip prefix-list inbound_routes seq 5 deny 2.128.0.0/16
17 route-map as2_to_as1 deny 1
18   match ip address prefix-list inbound_routes
```

Fig. 1. An example configuration update. The route map named as2_to_as1 is added on DCA_1 (lines 13–18) that denies route announcements sent from WAN_2.

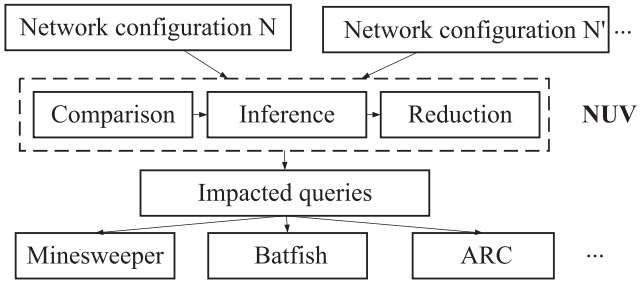


Fig. 2. High-level vision of NUV.

involves too much computation (i.e., the *global computation* with the network model.).

We propose a framework named NUV to address the above challenges. NUV outputs the impacted queries for a set of configuration updates, which can act as guidance for other verification tools (e.g., Batfish [4], ARC [5], ERA [6], and Minesweeper [1]). Fig. 2 shows a high-level picture of our framework, NUV. The basic steps in NUV are as follows: (i) Comparison. NUV compares the new configuration with the original configuration to obtain detailed updated configuration snippets. (ii) Inference. It preliminarily infers the potential impacted traffic based on a network instance that models the routing and forwarding behavior for each pair of endpoints. Specifically, by inspecting the configuration features and their effects on the route propagation process, NUV determines the range of impacted traffic. (iii) Reduction. It reduces the potentially

impacted traffic by removing items with *equivalent* forwarding behaviors. In summary, we make the following contributions.

- Determine the queries of the impacted traffic for configuration updates based on the network instance model. To the best of our knowledge, NUV is the first framework to determine queries for impacted traffic.
- We provide effective algorithms to preliminarily infer the impacted traffic of the updated configuration by exploring the effect of configuration snippets in the network model.
- We propose a variety of conditions to locally validate the equivalence of the forwarding behavior in the original and updated configuration for potential impacted traffic.
- We implement the above framework and evaluate it with a series of benchmark network configurations.

The remainder of this paper is organized as follows. Section 2 provides the definition of a network instance to facilitate subsequent presentation. Section 3 and Section 4 provide algorithms and conditions for determining the queries of the impacted traffic. The implementation and evaluation of NUV are provided in Section 5. Finally, Section 6 reviews related works, and Section 7 concludes the article.

2. Network model

To capture the effects of configuration on traffic forwarding, we introduce a network model and related preliminaries in this section. Specifically, we define the *network instance* per each traffic in the network to capture the effects of configuration on traffic forwarding. The traffic forwarding path is formalized as a *network instance solution*.

We give the insight behind our approach before the details of the network instance are described. For each traffic, our goal is to determine whether its forwarding behavior in the new configuration is the same as that in the original configuration. Intuitively, we can compute the forwarding behavior for the traffic with the original and updated network configurations; then, by comparing the obtained forwarding paths, we can determine whether the traffic is impacted. However, the computation in this approach is as expensive as that of verifying the corresponding query, which violates our goal of ruling out unimpacted traffic to reduce the verification cost. In the computation involved in verifying a single query, a network instance is constructed based on the input network configuration and a reachability query. Then, the verification tool computes the detailed forwarding path corresponding to the input query based on the network instance and returns the verification results based on the forwarding path. Specifically, to obtain the forwarding path, the tool performs *global computation* and handles protocol intersections on the possible paths based on the network instance. The global computation is PSPACE-complete [1,6,10].

Based on the above considerations, we propose a set of conditions that can be used to determine whether the traffic forwarding behavior in an updated network instance is the same as or equivalent to that in the original network instance. In this way, we do not need to perform global computation to obtain detailed forwarding paths. These conditions can be checked locally on changed topology or configuration snippets. We formally prove these conditions; the related introduction is elaborated in Section IV. If the network instance and its corresponding updated network instance satisfy these conditions, their final traffic forwarding behavior will be equivalent. Then, to determine whether the traffic is impacted, we can check whether its updated instance satisfies the local locations to avoid global computation. In fact, the approach can be improved further. Intuitively, if we add a specific ACL, only traffic matching the specific source and destination in the ACL configuration construct will be impacted. Formally speaking, the addition of an ACL does not impact other traffic network instances. It is determined by the ACL configuration construct does not affect the final traffic forwarding results of other traffic network instances. Therefore, we can analyze the configuration snippets' effects on network instances and preliminarily infer the potential impacted traffic based on that analysis. Subsequently, using local conditions, we can eliminate unimpacted traffic. We now describe the definition of network instance, as the analysis in the approach above is based on the exploration of the network instance.

Table 1
Definitions for network instance.

Network instance $\mathcal{N}_d^s = (V, I, \gamma, d, s, R, r_d, \psi, \eta, \prec)$			
V			nodes
I			ports
γ	:	$V.I \to V.I$	links
d	:	$V.I$	destination
s	:	$V.I$	source
$R_\perp$	=	$R \cup \{\perp\}$	route attributes or no attributes
$A_\perp$	=	$A \cup \{\perp\}$	packet attributes or no attributes
r_d	:	$R_\perp$	initial route attribute
ψ	:	$R \times V.I \to R$	control plane transfer function
η	:	$A \times V.I \to A$	ACL function
$\prec$	$\subseteq$	$R \times R$	selection function

2.1. Network instance

We provide our intuition that underlies network instances. Intuitively, without considering the function of a network, network topology is simply a graph in which the nodes are routers and the edges are the links between these routers. The network's control plane transforms routing messages using a collection of router-local rules defined by router configurations. Various routing protocols exist, and these protocols use different types of messages. For example, BGP messages include a destination prefix, an AS path, a local preference and so on. We introduce the route attribute, which captures a set of attributes of the routing messages exchanged by different protocols. The router-local rules defined by these configurations can be factored into two generic parts: (1) a control plane transfer function that transforms incoming and outgoing route attributes; and (2) a comparison relation that selects the best route attribute among the available route attributes to the same destination prefix. Although the router-local rules determine how packets should be forwarded in the control plane, the actual data forwarding behavior is impacted by ACLs, which can deny or permit traffic directly on the data plane; therefore, we introduce the ACL function. Next, we show the details of the terms used for network instances, and finally, we use a toy example to illustrate a network instance.

As shown in Table 1, a network instance $\mathcal{N}_d^s$ is a tuple ($V, I, \gamma, d, s, R, r_d, \psi, \eta, \prec$).

Definition (Graph): V, I and γ define the network graph, where V is a set of nodes, $V.I$ is the set of ports on those nodes, and γ defines the link connection relations between these ports. If $v.i = \gamma(v'.i')$, a link exists between port $v'.i'$ and port $v.i$. Here, d and s represent the destination and source ports, respectively, where $d \in V.I$ and $s \in V.I$. In addition, we use node(d) and node(s) to indicate the nodes to which the destination port and source port belong.

Definition (Route attributes): We use R to describe a set of routing message attributes exchanged by routing protocols on the control plane. For example, r.lp represents the BGP local preference value, $\perp$ denotes the absence of a routing message, and r_d represents the value of the initial routing message advertised by destination d. For each network instance, r.prefix and r.length are determined by the initial routing message value at the destination.

Definition (Control plane transfer function): ψ: $R \times V.I \to R$ is the function that models how the routing attributes are modified between routers. Given a port $v.i$ ($v.i \in V.I$) and the route attribute r ($r \in R$) from its neighbor $\gamma(v.i)$ across the link, the function determines which new route attribute is received and processed at the current node v. Based on the routers' configurations, it encodes control plane operations based on the routing attributes, including output filtering on the neighbor $\gamma(v.i)$, the current node's input filtering, route aggregation and so on. $\prec$ is a

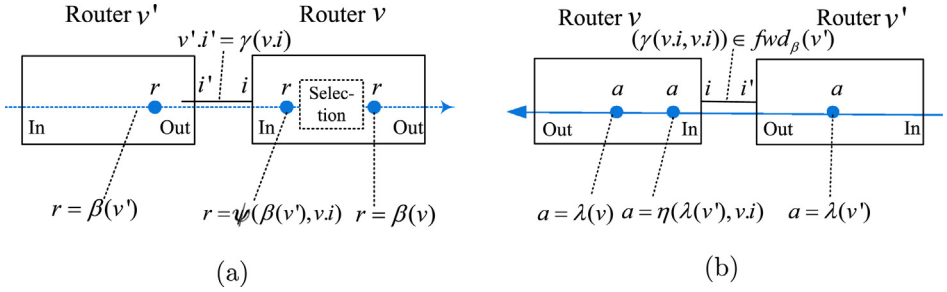


Fig. 3. An example of a network instance solution.

partial order of v used to select the best route attribute r out of a set of route attributes to the same destination prefix. If route attribute $r_1 \prec r_2$, then r_1 is more desirable. The partial order models the routing decision procedure that compares route attributes using some combination of the message fields in these attributes. Fig. 3(a) shows an example of the transfer function ψ when processing the route attribute r, and we elaborate on this function in Section 2.3.

Definition (Packet attributes): The network data plane takes as input a packet on a port and may move the (possibly modified) packet to another port based on the forwarding decision on each node. We use A to represent the packet attributes passing from source s to destination d on the data plane. A represents the values of various packet fields such as the source IP, destination IP, MAC address and port number. Specifically, a_s is the initial packet attribute at source s to reach destination d.

Definition (ACL function): The actual data plane forwarding behavior can also be restricted by the ACLs on each node. The ACL function η is provided to model how ACLs impact traffic forwarding: η: $A \times V.I \rightarrow A$. Given a port $v.i$ and the packet attribute a from neighbor $\gamma(v.i)$ across the link, if the packet attribute satisfies the match conditions of ACL on neighbor $\gamma(v.i)$ or current node's port $v.i$, the function modifies and produces new packet attributes according to the action of the ACL. Fig. 3(b) provides an example of the function η on processing the attribute a.

The generality of the network instance not only helps model a wide variety of routing protocols but also allows nonsensical behaviors. For example, with the definition in Table 1, it allows that there is a port that has a link to itself, while in a real network, there is no such a case. Therefore, we need to add practical constraints for the definition of network instances. A well-formed network instance satisfies the following constraints. (i) Self-loop-free. $\forall v.i.\ \gamma(v.i) \neq v.i,\ v.i \in V.I$. This means that no port has a link to itself.[1]. (ii) Nonspontaneous. $\forall v.i.\ \eta(\perp, v.i) \rightarrow \perp, v.i \in V.I$. A node does not obtain a route from that neighbor if its neighbors have no route to the destination (except static routing).

2.2. Network instance solution

We are now ready to formalize the network instance solution. The control plane solution β: $V \rightarrow R$ encodes the final route attribute selected to forward the traffic from source s to destination d. The data plane solution λ: $V \rightarrow A$ represents the final packet attribute on each node in the network instance. Given the data plane solution, users can check a wide range of network properties (e.g., reachability, black holes, forwarding loops) using a set of verification queries.

Definition (Control plane solution): Solution $\beta(v)$ is a route attribute labeling that maps node v to a final route chosen for forwarding the traffic.

$$\beta(v) = \begin{cases} r_d & v = \text{node}(d) \\ \perp & \text{attr}_\beta(v) = \phi \\ r \in \text{attr}_\beta(v) \text{ that is optimal by } \prec & \text{attr}_\beta(v) \neq \phi \end{cases}$$

$$\text{attr}_\beta(v) = \{r | (r, v.i, v'.i') \in \text{choices}_\beta(v)\}$$

$$\text{choices}_\beta(v) = \{(r, v.i, v'.i') | v'.i' = \gamma(v.i), r = \psi(\beta(v'), v.i), r \neq \perp\}$$

$\text{choices}_\beta(v)$ are the route choices available from neighbors. The values are not null if and only if some of their ports $v.i$ possess link connections with other neighbor ports $v'.i'$, and the attribute r ($r = \psi(\beta(v'), v.i)$) obtained by applying the transfer function ψ from the neighbor's label $\beta(v')$ is not null. $\text{attr}_\beta(v)$ represents the attributes in the available $\text{choices}_\beta(v)$ from the neighbors. For a solution $\beta(v)$, there are three cases to consider. (i) When the node is the destination d, $\beta(v)$ is the initial value of the route attribute r_d. (ii) The value is $\perp$ when no attributes $\text{attr}_\beta(v)$ are available from neighbors. (iii) The solution on node v is the winner among $\text{attr}_\beta(v)$ by applying a selection process $\prec$.

$$\text{fwd}_\beta(v) = \{(v.i, v'.i') | (r, v.i, v'.i') \in \text{choices}_\beta(v), r = \beta(v)\}$$

$\text{fwd}_\beta(v)$ are the edges used to identify the best choice among the neighbors.

As shown in Fig. 3(a), after $\beta(v')$ goes out of node V' and into node V, the route attribute at node V becomes $\psi(\beta(v'), v.i)$. Considering that node V can receive more than one route attribute for the same destination prefix, $\prec$ helps determine the best attribute as $\beta(v)$.

Definition (Data plane solution): Solution $\lambda(v)$ is a traffic attribute labeling that maps node v to the final forwarding packet attribute.

$$\lambda(v) = \begin{cases} a_s & v = \text{node}(s) \\ \{a | (a, v.i, v'.i') \in \text{choices}_\lambda(v)\} & v \neq \text{node}(s) \end{cases}$$

$$\text{choices}_\lambda(v) = \{(a, v.i, v'.i') | v'.i' = \gamma(v.i), (v'.i', v.i) \in \text{fwd}_\beta(v'), a = \eta(\lambda(v'), v.i)\}$$

$\text{choices}_\lambda(v)$ are the forwarding choices available from its neighbors. $(v'.i', v.i) \in \text{fwd}_\beta(v')$ are the edges used to learn the best route attribute choice for the neighbor v'. These edges also imply the potential paths used to forward traffic. To obtain $\text{choices}_\lambda(v)$, we apply the function η on its neighbor's label $\lambda(v')$ based on the output filtering ACL of the neighbor $\gamma(v.i)$ and the input filtering ACL of port $v.i$. For solution $\lambda(v)$, if node v is the source node, it takes the initial value a_s. Otherwise, it is an attribute in the available $\text{choices}_\lambda(v)$ among its neighbors.

$$\text{fwd}_\lambda(v) = \begin{cases} \perp & v = \text{node}(s) \text{ or } \{v \neq \text{node}(s), \lambda(v) = \perp\} \\ \{(\text{fwd}_\lambda(v'), (v.i, v'.i')) | a \in \lambda(v), & \\ a \neq \perp, (a, v.i, v'.i') \in \text{choices}_\lambda(v)\} & v \neq \text{node}(s), \lambda(v) \neq \perp \end{cases}$$

We use $\text{fwd}_\lambda(\text{node}(d))$ to denote the final traffic forwarding paths in the network instance. For node(s), the packet has not been forwarded; thus, $\text{fwd}_\lambda(\text{node}(s))$ is $\perp$. If the node is not the source but $\lambda(v)$ equals $\perp$, it means that the packet has reached the current node; thus, $\text{fwd}_\lambda(v)$ is also $\perp$. Otherwise, edge $(v.i, v'.i')$ that forwards the best choice a ($a \in \lambda(v)$) is added to the edge set $\text{fwd}_\lambda(v')$.

[1] Note that self-loop-free is different from another term, *loop-free forwarding*. The loop-free forwarding property is typically referred to as a network property that must be verified, which means that no packet should exist that would cause a forwarding loop in the network.

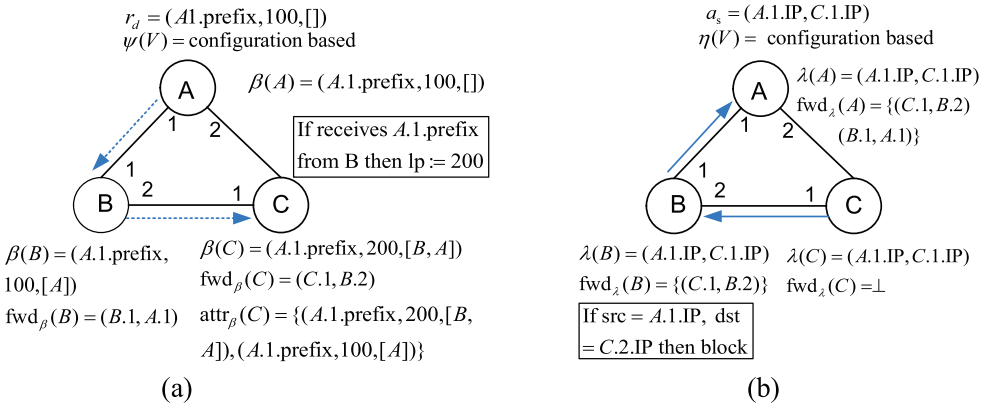


Fig. 4. An example of network instance $\mathcal{N}^{C.1}_{A.1}$. For routers B and C, the configuration-based parts of the transfer function and the ACL function are shown in the boxes.

In Fig. 3(b), $\lambda(v')$ exits from node V' and arrives at node V. The attribute at the incoming port of node V is $\eta(\lambda(v'), v.i)$. If there is more than one choice in $\lambda(v)$, then node v may forward the packet to multiple neighbors.

Fig. 4 shows a toy example of a network instance $\mathcal{N}^{C.1}_{A.1}$. The route attribute r is simplified to (r.prefix, r.lp, r.path), where r.lp is the local preference and r.path represents the AS path. The control plane transfer function $\psi(v)$ on each router is configuration based. In this network, router C sets the local preference of the received routing message from router B to 200. The control plane solution on each router $\beta(v)$ is shown in Fig. 4. Specifically, we show the values of $\text{attr}_\beta(C)$, and for $\beta(C)$, the node in $\text{attr}_\beta(C)$ with the highest local preference is the winner. The ACL function $\eta(v)$ is also configuration based. $\eta(B)$ blocks traffic whose source is $C.2$ and whose destination is $A.1$. For this network instance $\mathcal{N}^{C.1}_{A.1}$, $\eta(B)$ does not block the traffic from $C.1$ to $A.1$. The data plane solution on each router $\lambda(v)$ is shown in the figure. The dotted and solid arrows indicate the final routing and forwarding paths of this network instance, respectively.

Network query: Users can check the reachability query for the specific traffic based on the data plane solution λ of the corresponding network instance $\mathcal{N}^{s}_{d}$. Traffic from source s can reach destination d, if a path $\text{fwd}_\lambda(\text{node}(d))$ $(x_1, x_2, \ldots, x_k)$ exists where the network forwards along the path using the traffic labels $\lambda(x_1), \lambda(x_2), \ldots, \lambda(x_k)$, $x_1 = node(s)$ and $x_k = node(d)$. For the toy example in Fig. 4, $\text{fwd}_\lambda(A)$ is ($C.1$, $B.2$, $B.1$, $A.1$); thus the reachability query (i.e., Can traffic from $C.1$ reaches $A.1$?) will pass the verification test. Similarly, the bounded path length query checks whether the final forwarding path is less than a certain length m. Traffic from source s can reach destination d with a certain length m, if a forwarding path $\text{fwd}_\lambda(\text{node}(d))$ $(x_1, x_2, \ldots, x_k)$ exists where $x_1 = \text{node}(s)$ and $x_k = \text{node}(d)$, and whose path is less than m. Note that, like other verification tools, after the forwarding behavior of the traffic is well modeled and computed, given the data plane solution λ, NUV can be used to verify the reachability query. However, NUV aims to address the lack of knowledge problem and speed up the verification of network configuration updates. NUV does not focus on verifying reachability or performing any other queries; instead, it outputs the impacted queries caused by the configuration updates.

2.3. Control plane transfer function

From the above introduction to a network instance, we know that the control plane transfer function is configuration-based and that configuration updates can impact the function. In this subsection, we show how the various configurations impact the processing of route attributes in the transfer function.

Algorithm 1 provides the subprocedures in the control plane transfer function ψ, where cs in line 1 is the router configuration corresponding to port $v.i$ and its neighbor $\gamma(v.i)$. It consists of the following steps. (i) *Output filtering*: Given the route attributes r of the current

Algorithm 1: Transfer Function $\psi(r, v.i)$.

```
1  Suppose cs = (fo[.], fi[.], cr[.], ca[.], sr[.]) ;
2  for i = 1 to SIZE(fo[.]) do
3      if r.prefix matches fo[i].condition then
4          r ←UPDATE(r, fo[i].action)            ▷ Output filtering on γ(v.i).
5  for i = 1 to SIZE(fi[.]) do
6      if r.prefix matches fi[i].condition then
7          r ←UPDATE(r, fi[i].action)            ▷ Input filtering on v.i.
8  for i = 1 to SIZE(cr[.]) do
9      if r.proto equals cr[i].fromProto then
10         r ←UPDATE(r, cr[i])                   ▷ Route redistribution.
11 for i = 1 to SIZE(ca[.]) do
12     if r.prefix matches ca[i].perfix then
13         r ←UPDATE(r, ca[i])                   ▷ Route aggregation.
14 for i = 1 to SIZE(sr[.]) do
15     if r.prefix matches sr[i].prefix then
16         r ←UPDATE(r, sr[i])                   ▷ Static routing.
```

node's upstream neighbor, the output filters fo[.] (e.g., route map) on its connected neighbor $\gamma(v.i)$ are applied to modify or drop the matched route advertisements r according to its action fo[i].action (lines 2–4). (ii) *Input filtering*: The input filters fi[.] on port $v.i$ are similarly applied to modify the incoming route attributes (lines 5–7). (iii) *Route redistribution*: This step captures cross-protocol interactions if the r.proto attribute is equal to the original protocol cr[i].fromProto of the redistribution configuration cr[i] (lines 8–10). By default, it can work on route attributes with any prefix. (iv) *Route aggregation*: This step is applied to announce a less-specific prefix that covers many, more-specific prefixes (lines 11–13). An aggregate route in configuration is activated only if the device receives an input route that is more specific than the aggregate route ca[i].prefix. (v) *Static route*: If the prefix r.prefix matches the prefix of static route sr[i].prefix, the static route takes precedence over any dynamic routes computed by routing protocols (lines 14–16).

When multiple available routes exist to the destination, the selection function $\prec$ determines best route for the prefix; the comparison relation can utilize any protocol configurations proto[.] on the node, including routing processes, routing adjacencies, link cost or any other protocol-specific attributes. Specifically, if the routes belong to the same protocol, the protocol is used to determine the best routing; otherwise, the route with the lowest administrative distance (AD) value will be selected.

3. Network instance inferences

Given the above introduction, we know that if the data plane solution of the network instance has not changed, the verification result

of the corresponding query will remain the same. For each network instance, the data plane solution depends on the control plane transfer function, ACL function, and the selection function on each router, and these functions are configuration-based. By analyzing the changed configurations and their impacts on these functions, we can determine whether the changed configuration will alter the output of the transfer function, ACL function and selection function on each router for each network instance. In this section, we show how to preliminarily infer the impacted network instances where the output of the transfer function, ACL function or selection function will be altered because of the configuration changes.

3.1. Inference

Given the updated network configuration, it is easy to obtain the updated network instance $\mathcal{N}'^{s}_{d}$ for traffic from source s to destination d. By introducing the data plane solution, we know that the network properties to be verified are encoded as constraints on the data plane solution of the network instance. If the existing data plane solution $\lambda(v)$ is equal to the updated solution $\lambda(v)'$ to $\mathcal{N}'^{s}_{d}$, we know that the corresponding queries do not need to be rechecked. Otherwise, the queries related to the data plane solution $\lambda(v)$ will be impacted by the configuration updates.

We now explain how to check whether the data plane solution $\lambda(v)$ will be impacted by the configuration updates. The transfer function ψ, ACL function η and selection function $\prec$ in a network instance are configuration-based. Router configuration updates will impact these functions. Let ψ_v and ψ'_v denote the original and updated function ψ of node v, respectively. For instance, $\mathcal{N}^{s}_{d}$ and updated instance $\mathcal{N}'^{s}_{d}$, if $\psi_v(r, v.i) = \psi'_v$ $(r, v.i)$, where $r \in R$, then the data plane solution λ equals λ' based on the definition of data plane solution; otherwise, the data plane solution λ may be impacted. The updated function η' can be similarly handled.

To check whether $\psi_v(r, v.i)$ is equal to $\psi'_v(r, v.i)$, we need to explore the updated configuration snippets and their corresponding impacts on the transfer function ψ. Algorithm 1 captures the effects of various configuration snippets on the processing route attribute r. Based on this algorithm, we can explore the updated configuration snippets and their corresponding changes to the transfer function ψ. We adopt the addition of the route map $as2_to_as1$ in Fig. 1 as an example. After the configuration updates, the following process logic of the input filter is added to function ψ.

$$\text{if}(r.\text{prefix} = 2.128.0.0) \wedge (r.\text{length} = 16)$$
$$r.\text{valid} = \text{false}.$$

This addition means that if the prefix in route attribute r matches the first 16 bits of 2.128.0.0, the transfer function drops this route attribute. Now, we explore the execution code paths of the input route attribute r in the original transfer function ψ and the updated transfer function ψ' for router DCA_1. For function ψ, the input r will never execute lines 5, 6 and 7 of Algorithm 1 because the input filter fi[.] of DCA_1 is null. In the updated function ψ', if the input r satisfies the match condition of the input filter, r will pass through a different execution code path in ψ' compared with the code path in ψ. Otherwise, r will pass through the same code as the code path in ψ. For an r that satisfies the condition $(r.\text{prefix} = 2.128.0.0) \wedge (r.\text{length} = 16)$, the output of the transfer function ψ will be updated; thus, we know that $\psi_v(r, v.i) \neq \psi'_v(r, v.i)$. According to the definition of the network instance, for each $\mathcal{N}^{s}_{d}$, r.prefix and r.length are determined by the initial values at its destination d. Therefore, network instances whose destination route attributes match the prefixes in the configuration will be impacted: $\{\mathcal{N}^{s}_{d} | d.\text{ip} \in \text{fbm}(\text{fi.prefix}, \text{fi.length}), s \in V.I, d \in V.I\}$, where fi is the updated filter, and fbm encodes the IP, which can be matched by the prefix fi.prefix with the length fi.length. Other updated configuration snippets can be handled similarly.

By analyzing the transfer function, we can determine what inputs cause each part of the function to execute, and by following a symbolic execution method, we can logically explore all the execution paths of the transfer function. For each execution path, we can collect the match conditions (e.g., the condition $(r.\text{prefix} = 2.128.0.0) \wedge (r.\text{length} = 16)$ in the above example), and then determine which configuration constructs will impact which match conditions. In this way, we can determine how configuration construct updates impact the conditions and the execution paths in the transfer function. As the above toy example shows, we can then determine what inputs will be impacted because they will follow a new execution path. In this way, we can obtain the impacts of various configuration snippets, which can be classified into the following four types, as shown in Table 2 and Table 3: (I) Network instances with specific sources and destinations. Because the ACL function restricts specific traffic on the data plane, instances whose source and destination match the updated ACL configuration (denoted as al) will be impacted. (II) Multiple network instances with a specific destination. The impacted instances are matched by the updated input filter fi, static router sr, prefix announcement protoc, output filter fo, etc. protoc refers to the configuration that announces routes using BGP or any other routing protocols with a network prefix. In addition, if route redistribution cra propagates routing information from a static route into another protocol, the network instances whose destination equals the destination IP address in the static route configuration will be impacted. (III) All network instances except those with intradomain sources and destinations. By default, a configuration update to any routing protocol-specific attributes (e.g., updating routing adjacencies) will apply to all network instances. When we consider the *route propagation process*, changing certain attributes of interdomain routing processes (e.g., removing eBGP routing adjacencies protoa) will not impact the traffic from the intradomain source to the intradomain destination. We use intra($V.I$) to denote intradomain endpoints, where s and d are in the same AS. (IV) All network instances. As previously introduced, updating an intradomain routing adjacency protob will impact the forwarding behaviors of all network instances.

3.2. Computation

Based on the above observations, we infer the potentially impacted network instances using the following steps (see Algorithm 2). By processing the configuration c[.] and the updated configuration c'[.], we can obtain the detailed updated configuration snippets cc[.] and their type (denoted as flag) (line 1). Note that the sources or destinations in the network can actually be partitioned into traffic equivalence classes

Algorithm 2: Inference Function $Infer(c[.], c'[.])$.

```
1  (cc[.], flag) ← COMPARE(c[.], c'[.]) ;
2  ec[.] ← CLASSIFY(c[.], c'[.])          ▷ ec[.] are the traffic classes.;
3  if flag matches configuration type protob or crb then
4      n[.] ← GETALL(ec[.]) Break
5  else if flag matches configuration type protoa then
6      n[.] ← GETINTER(ec[.])
7  else if flag matches configuration type fi, cra, sr, protoc or fo then
8      pf[.] = GETPREFIX(cc[i], flag);
9      n[.] ← n[.] ∪ GETMUL(ec[.], pf[.])
10 else if flag matches configuration type al then
11     sd[.] = GETSRCDST(cc[i], flag);
12     n[.] ← n[.] ∪ GETSPE(ec[.], sd[.])
13 for n ∈ n[.] do
14     if EQUAL(n, n', c, c') then
15         n[.] ← REDUCE(n, n[.])
```

Table 2
Sub-procedures and corresponding configurations.

Sub-procedures	Configuration (logogram)		Type
Input filtering	Route map (fi)		II
Route redistribution	Redistribution	Static route to others (cra)	II
		BGP to others, etc. (crb)	IV
Route aggregation	Route aggregation (ca)		II
Static route	Static route (sr)		II
Output filtering	Route map (fo)		II
Route selection	BGP, OSPF, etc.	Interdomain routing adjacencies (protoa)	III
		Intradomain routing adjacencies, etc. (protob)	IV
Route stemming	Prefix announcement in protocols (protoc)		II
Data plane filtering	Access control list (al)		I

Table 3
Impacted network instances.

Type	Network instances
I	$\{\mathcal{N}_{\mathrm{d}}^{\mathrm{s}} \mid s.\mathrm{ip} \in \mathrm{al.srcip}, d.\mathrm{ip} \in \mathrm{al.dstip}, s \in V.I, d \in V.I\}$
II	$\{\mathcal{N}_{\mathrm{d}}^{\mathrm{s}} \mid d.ip \in \mathrm{fbm(tp.prefix, tp.length)}, \mathrm{tp} \in \{\mathrm{fi, cra, ca, protoc, sr, fo}\}, s \in V.I, d \in V.I\}$
III	$\{\mathcal{N}_{\mathrm{d}}^{\mathrm{s}} \mid s \notin \mathrm{intra}(V.I) \vee d \notin \mathrm{intra}(V.I), s \in V.I, d \in V.I\}$
IV	$\{\mathcal{N}_{\mathrm{d}}^{\mathrm{s}} \mid s \in V.I, d \in V.I\}$

where each class has a collection of IP ranges [5].[2] This allows us to build network instances that address per class to per class traffic instead of per address to per address traffic.

The algorithm obtains the equivalence classes ec[.] by locating any prefix in the configurations in [5] (line 2). If updated constructs exist such as a route redistribution from BGP to others crb, via the GetAll function, we know that the impacted instances n[.] are all possible network instances in the network (lines 3–4). Otherwise, if interdomain protocol configurations proboa exist, we add the impacted instances of type III to n[.] through the GetInter function (lines 5–6). Next, we account for the configuration construct updates, including input filters fi, route aggregation ca, static routes sr, output filters fo and so on. After processing the related destination prefixes pf[.] in these updates, we can determine the impacted network instances with these specific destinations through the GetMul function (lines 7–9). Similarly, we can accommodate ACL updates by collecting the specific source and destination sd[.] in the updated configuration (lines 10–12). Then, we obtain the potential impacted network instances where the output of transfer functions or the ACL function will be altered. In fact, for some instances, the updated network instance will still share the same or similar forwarding behaviors as the original instance. Thus, to obtain the final result, we also need to remove these network instances (lines 13–15),. We provide detailed solutions to this problem in the following section. Based on this inference processing approach, the number of impacted network instances may depend on the number of updated configuration snippets and types. When updated configuration snippets of type IV exist, the number of impacted instances could reach n^2, where n is the number of traffic classes. Otherwise, the number of impacted network instances is $\mathcal{O}(n^2 - m^2 + v * n + k)$, where m is the number of intradomain endpoints, v is the number of configuration snippets of type II-like static routes, and k is the number of specific source and destination pairs in the updated ACL. The complexity of comparing configurations is $\mathcal{O}(l * d + p * q)$, where l is the maximum number of routing processes (e.g., OSPF, RIP, BGP) running on a single router, d is the maximum number of devices participating in a single routing process, p is the total number of ACLs and router map filters, and q is the maximum number of lists in each ACL or route map. Inferencing depends on checking the updated configuration snippets, and it iterates over these constructs to determine the impacted network instances, thus yielding a complexity of $\mathcal{O}(v + k)$.

[2] ARC [5] can also leverage the traffic equivalence classes to reduce the amount of traffic that must be analyzed.

4. Network instances reduction

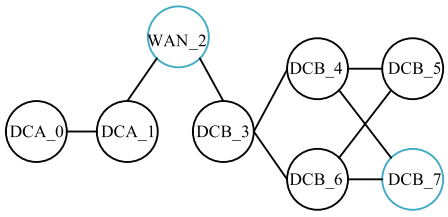
For the potential impacted network instances in the outputs of inference, the outputs of some transfer functions and ACL functions on routers will change because of the changed configurations. In fact, for each network instance, even when the output of some transfer functions and ACL functions on routers changes, the updated data plane solution may still be the same or similar to the original data plane solution. In these cases, the verification result of the corresponding query will remain unchanged. In this section, we explore the conditions that cause existing network instances and updated instances to share equivalent forwarding behaviors.

4.1. Network subinstance

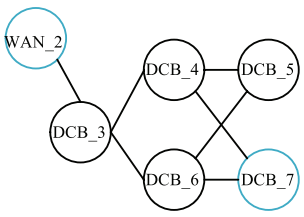
Considering the addition of route map $as2_to_as1$ in Fig. 1, according to inference, the output of the transfer function ψ in network instances whose destination is WAN_2 will change. Intuitively, we expect that certain network instances whose destination is WAN_2 will be impacted; however, in fact, traffic from DCB_7 to WAN_2 will not be impacted. Recalling the loop-free and nonspontaneous properties of the network instance, a destination prefix that arrives at DCB_7 will never pass through the interface of as2_to_as1 on DCA_1. The added route map is not on the routing advertisement's paths for the destination DCB_7 prefix. In other words, the route map update does not occur at the nodes in the *network subinstance* in Fig. 5(b) for the traffic from DCB_7 and WAN_2, thus the existing network subinstance and its updated network subinstance share the same forwarding behavior. We provide a detailed explanation below.

Formally, our goal is to determine the conditions when $\psi_v(r, v.i) \neq \psi'_v(r, v.i)$ or $\eta_v(a, v.i) \neq \eta'_v(a, v.i)$, and the data plane solution λ' for $\mathcal{N}'^{\mathrm{s}}_{\mathrm{d}}$ are equivalent to λ for $\mathcal{N}^{\mathrm{s}}_{\mathrm{d}}$ (where v is the updated node). In this subsection, we propose the concept of a *network subinstance*, in which traffic forwarding in the network subinstance is the same as that in the original network instance. Given a network instance, we can obtain its network subinstances through various forms of transformation. If the network subinstance of $\mathcal{N}^{\mathrm{s}}_{\mathrm{d}}$ is the same as the network subinstance of $\mathcal{N}'^{\mathrm{s}}_{\mathrm{d}}$, then λ is equal to λ'.

Definition (Network subinstance): Network subinstance $\mathcal{S}|\mathcal{N}^{\mathrm{s}}_{\mathrm{d}}$ $(\overline{V}, \overline{I}, \overline{\gamma}, \overline{d}, \overline{s}, \overline{R}, \overline{r_{\mathrm{d}}}, \overline{\psi}, \overline{\eta}, \overline{\prec})$ is effective for network instance $\mathcal{N}^{\mathrm{s}}_{\mathrm{d}}$ $(V, I, \gamma, d, s, R, r_{\mathrm{d}}, \psi, \eta, \prec)$ if the following conditions hold: (i) *dst-equivalence*. $d = \overline{d}$. (ii) org-equivalence. $r_{\mathrm{d}} = \overline{r_{\mathrm{d}}}$. (iii) src-equivalence. $s = \overline{s}$. (iv) trans-equivalence. $\forall r \in R, v.i \in V.I, \overline{r} \in \overline{R}, \overline{v.i} \in \overline{V.I}$. $\psi(r, v.i) =$



(a) Topology for $\mathcal{N}^{DCB_7}_{WAN_2}$.



(b) Topology for $\mathcal{S}|\mathcal{N}^{DCB_7}_{WAN_2}$.

Fig. 5. Topologies of network instances for traffic from DCB_7 to WAN_2. In (b), the links and nodes that are not on the routing advertisement's paths from WAN_2 to DCB_7 are removed.

$\overline{\psi}(\overline{r}, \overline{v.i})$ (v) acl-equivalence. $\forall a \in A, v.i \in V.I, \overline{a} \in \overline{A}, \overline{v.i} \in \overline{V.I}.\ \eta(a, v.i) = \overline{\eta}(\overline{a}, \overline{v.i})$ (vi) select equivalence. $\forall r \in R, r' \in R, \overline{r} \in \overline{R}, \overline{r'} \in \overline{R}.\ r \prec r' \Leftrightarrow \overline{r} \prec \overline{r'}$ (vii) Slicing. $\forall v.i \in V.I, \overline{v.i} \in \overline{V.I}.(v'.i' \in \gamma(v.i)) \wedge (v.i, v'.i' \in p) \Leftrightarrow \overline{v'.i'} \in \gamma(\overline{v.i})$, where $p = \{x_0, x_1, \ldots, x_i, \ldots, x_j, \ldots, x_n\}$, $x_0 = d$, $x_n = s$, $x_i \in V.I$ and $x_i \neq x_j$. p are the loop-free paths from destination node(d) to source node(s).

Theorem 1. *Suppose we have a well-formed $\mathcal{N}^s_d$, $\mathcal{S}|\mathcal{N}^s_d$. For any solution λ of $\mathcal{N}^s_d$, there exists an equivalent solution λ' of $\mathcal{S}|\mathcal{N}^s_d$ and vice versa.*

Proof. ($\Rightarrow$) Given solution λ to $\mathcal{N}^s_d$, we show that when removing ports that are not in path p, solution λ will not be impacted. For solution λ, a path $p' = \text{node}(s), v_1, v_2, v_3, \ldots, v_i, \ldots, \text{node}(d)$ exists where $p' \in \text{fwd}_\lambda(d)$. The path label is $\lambda(\text{node}(s))$, $\lambda(v_1)$, $\lambda(v_2)$, $\lambda(v_3), \ldots, \lambda(v_i), \ldots, \lambda$ (node(d)). Assume solution λ is impacted; then, we know that a $\lambda(v_i)$, $v_i \notin p$ exists in the solution. Consider an arbitrary such v_i. By the definitions of λ and choices$_\lambda$, we know that $v_i.i,\ v_i.i \in \text{fwd}_\beta(v')$ exists. From the definitions of fwd$_\beta$ and choices$_\beta$, we know that $\lambda(v')$, $v'.i' = \gamma(v_i.i)$ exists. By the transitivity property of link connections, we know that solution λ forwards along a path $s, \ldots, v_i.i, v'.i', \ldots, d$. Considering loop-free and nonspontaneous properties of network instance, the path must be of one the loop-free paths from destination d to source s. This leads to a contradiction with the fact that some nodes in the solution λ exist that are not in p; hence, solution λ will not be impacted. Therefore, we have that an equivalent solution λ' of $\mathcal{S}|\mathcal{N}^s_d$ exists in solution λ.

($\Leftarrow$) We are given a solution λ' of $\mathcal{S}|\mathcal{N}^s_d$ and show that λ in $\mathcal{N}^s_d$ exists. Suppose λ' is $\lambda'(v_1), \lambda'(v_2), \lambda'(v_3), \ldots, \lambda'(v_i), \ldots, \lambda'(v_n)$, where $v_1 = \text{node}(s)$ and $v_n = \text{node}(d)$. Then, we show that adding by some ports or links to path p, the solution λ' of $\mathcal{S}|\mathcal{N}^s_d$ will not be affected. For example, if the solution λ' is impacted then we know that $\exists \lambda''(v_i).\ v_i \notin p$ in the solution λ''. Consider an arbitrary such v_i. By the definitions of λ'' and choices$_{\lambda''}$, we obtain $\exists v_i.i.\ v_i.i \in \text{fwd}_\beta(v')$. From the definitions of fwd$_\beta$ and choices$_\beta$, we know that $\exists \lambda''(v').\ v'.i' = \gamma(v_i.i)$. By the transitivity of link connection, we know that solution λ'' forwards along a path $s, \ldots, v_i.i, v'.i', \ldots, d$. Considering the loop-free and nonspontaneous properties of network instances, the path must be one of the loop-free paths. This contradicts the fact that some node exist in the solution λ'' that are not in the path p. Therefore, λ'' does not exist, and consequently, the solution λ' of $\mathcal{N}^s_d$ will not be affected. Considering the definition of $\mathcal{S}|\mathcal{N}^s_d$ and $\mathcal{N}^s_d$, we know that an equivalent solution exists: λ with solution λ'. □

Corollary 1. *Suppose a well-formed $\mathcal{N}^s_d$, $\mathcal{N'}^s_d$ is obtained by pruning $\mathcal{N}^s_d$ (i.e., removing some of the ports that are not in $\mathcal{S}|\mathcal{N}^s_d$). Then, for any solution λ' of $\mathcal{N'}^s_d$, equivalent λ of $\mathcal{N}^s_d$ exist.*

The proof is provided in the appendix. In addition, when type II configuration updates (e.g., adding a route map or removing a static route) do not occur at the nodes in the network subinstance, the solution of the network instance will not be affected. This can be proven in a similar way.

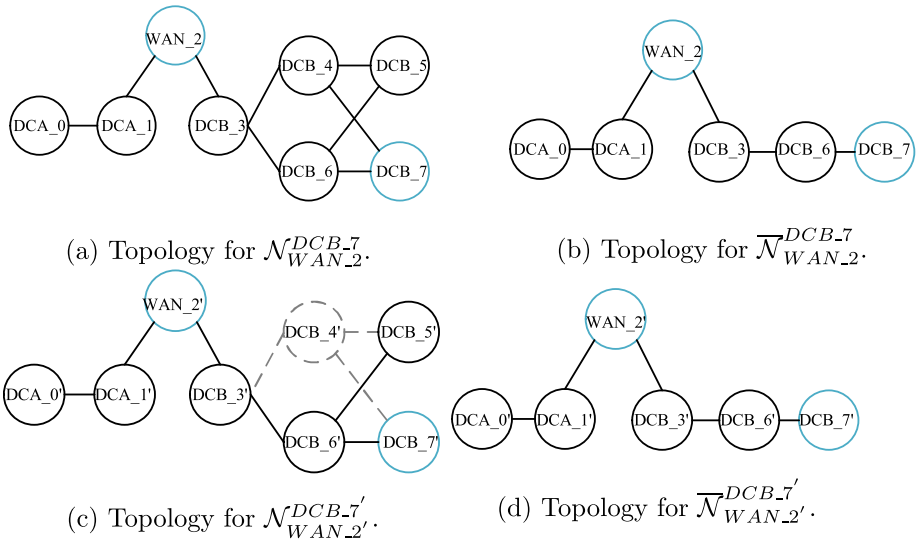
4.2. Abstract network instance

In some cases, there exists a node v for $\mathcal{N}^s_d$ and $\mathcal{N'}^s_d$ where $\psi_v(r, v.i) \neq \psi'_v(r, v.i)$ or $\eta_v(a, v.i) \neq \eta'_v(a, v.i)$. In these situations, the data plane solution λ' for $\mathcal{N'}^s_d$ is not equal to λ for $\mathcal{N}^s_d$, but a wide variety of network properties of λ for $\mathcal{N}^s_d$ are equivalent to λ' for $\mathcal{N'}^s_d$, and the verification results of the corresponding queries for $\mathcal{N}^s_d$ and $\mathcal{N'}^s_d$, such as reachability, loop freedom, and path length, are still the same. In this subsection, we propose *an abstract network instance* inspired by network control plane abstraction [9]. The traffic forwarding in the abstract network instance is equivalent to that in the original network instance (i.e., bisimulation). If the abstract instance of $\mathcal{N}^s_d$ is the same as the abstract instance of $\mathcal{N'}^s_d$, the verification result of the corresponding queries for $\mathcal{N}^s_d$ and $\mathcal{N'}^s_d$ will be the same. Considering the network in Fig. 6, suppose that DCB_4 is the backup router of DCB_6, and they share the same transfer and ACL functions. If DCB_4 is removed, all the network instances will be impacted according to the inference approach in Section III. However, the traffic from DCB_7 to WAN_2 is not impacted because its abstract network instance (Fig. 6(b)) and the abstract network instance under the updated configuration (Fig. 6(d)) are equivalent. According to the following definition of abstract instance, nodes with the same transfer function and ACL function (e.g., DCB_4 and DCB_6) will be combined in the abstract network instance. We provide a detailed explanation below.

Definition (Abstract network instance): We use the topology function f and the attribute function h to describe the relation of a concrete instance $\mathcal{N}^s_d$ (V, I, γ, d, s, R, r_d, ψ, η, $\prec$) and its abstract instance $\overline{\mathcal{N}}^s_d$ ($\overline{V}$, $\overline{I}$, $\overline{\gamma}$, $\overline{d}$, $\overline{s}$, $\overline{R}$, $\overline{r_d}$, $\overline{\psi}$, $\overline{\eta}$, $\overline{\prec}$). $f: f(v.i) = \overline{v.i}$ maps each concrete port to an abstract port, and $h: h(a) = \overline{a}$ maps each concrete attribute to an abstract port. The effective abstract network instance satisfies the following conditions. (i) dst-equivalence: $(\overline{d} = f(d)) \wedge (\forall\ d'.\ d \neq d' \Rightarrow f(d') \neq \overline{d})$. (ii) org-equivalence: $h(r_d) = \overline{r_d}$. (iii) src-equivalence: $(\overline{s} = f(s)) \wedge (\forall s'.\ s \neq s' \Rightarrow f(s') \neq \overline{s})$. (iv) trans-equivalence: $\forall r \in R, v.i \in V.I.\ h(\psi(r, v.i)) = \overline{\psi}(h(r), f(v.i))$ (v) acl-equivalence: $\forall a \in A, v.i \in V.I.\ h(\eta(a, v.i)) = \overline{\eta}(h(a), f(v.i))$ (vi) select equivalence: $\forall r \in R, r' \in R', \overline{r} \in \overline{R}, \overline{r'} \in \overline{R'}.\ r \prec r' \Leftrightarrow \overline{r} \prec \overline{r'}$ (vii) abstraction: $\forall v.i \in V.I.\ v'.i' \in \gamma(v.i) \Rightarrow \overline{v'.i'} \in \gamma(\overline{v.i})$, and $\forall \overline{v.i} \in \overline{V.I}.\ \overline{v'.i'} \in \gamma(\overline{v.i}) \Rightarrow (\forall v.i.\overline{v.i} = f(v.i) \Rightarrow \exists v'.i'.\overline{v'.i'} = f(v'.i') \wedge v'.i' \in \gamma(v.i))$. This means that for every concrete edge $(v.i, v'.i')$, there is a corresponding abstract edge $(\overline{v.i}, \overline{v'.i'})$, and for every abstract edge $(\overline{v.i}, \overline{v'.i'})$, the corresponding concrete port $v.i$ must have an edge to some concrete port $v'.i'$.[3] λ and $\overline{\lambda}$ are label-equivalent when $\forall v.\ h(\lambda(v)) = \overline{\lambda}(f(v))$. λ and $\overline{\lambda}$ are fwd-equivalence when $\forall v.i, v'.i'.\ (v.i, v'.i') \in \text{fwd}_\lambda(v) \Leftrightarrow (\overline{v.i}, \overline{v'.i'}) \in \overline{\text{fwd}}_{\overline{\lambda}}(\overline{v})$.

Theorem 2. *There is a solution λ to the well-formed network instance $\mathcal{N}^s_d$ if there is a solution $\overline{\lambda}$ to its effective abstract network instance $\overline{\mathcal{N}}^s_d$, and the solutions λ and $\overline{\lambda}$ are label-equivalent.*

[3] For BGP, another revised condition is needed [9].



(a) Topology for $\mathcal{N}_{WAN_2}^{DCB_7}$. (b) Topology for $\overline{\mathcal{N}}_{WAN_2}^{DCB_7}$.

(c) Topology for $\mathcal{N}_{WAN_2'}^{DCB_7'}$. (d) Topology for $\overline{\mathcal{N}}_{WAN_2'}^{DCB_7'}$.

Fig. 6. Topologies of network instances and abstract network instances for traffic from DCB_7 to WAN_2. The dotted line indicates the removal of DCB_4.

Proof. By the definition of λ, there are two cases to consider. First, we observe that if $node(v.i) = s$, then $\lambda(s) = a_s$. It follows that $\overline{\lambda}(f(s)) = \overline{\lambda}(\overline{s}) = \overline{a_s} = h(a_s)) = h(\lambda(s))$. For the second case, we show the implications separately.

($\Rightarrow$) We are given solution λ and show that there exists a label-equivalence solution $\overline{\lambda}$. By ACL-equivalence, we know that $\overline{\eta}(f(v.i), h(\lambda(v))) = h\ (\eta(v.i,\ \lambda(v))) = h(a)$, and by transitivity and label-equivalence (IH), we know that $\overline{\eta}(f(v.i), \overline{\lambda}\ (f(v)))) = h(a)$. By the lemma from [9], there is a solution β to $\mathcal{N}_{\mathrm{d}}^{\mathrm{s}}$ if there is a solution $\overline{\beta}$ to $\overline{\mathcal{N}}_{\mathrm{d}}^{\mathrm{s}}$, and β and $\overline{\beta}$ have control plane label-equivalence and fwd-equivalence. Hence, we obtain that $\forall v.i, v'.i', (v.i, v'.i') \in \mathrm{fwd}_{\beta}(v) \Leftrightarrow (\overline{v.i}, \overline{v'.i'}) \in \overline{\mathrm{fwd}}_{\overline{\beta}}(f(v))$. By the definition of $\mathrm{choice}_{\lambda}$, we know that $(h(a), f(v.i), f(v.i')) \in \overline{\mathrm{choice}}_{\overline{\lambda}}(f(v))$. Therefore, we have data plane choice equivalence. From Lemma 2, it follows that the solutions λ and $\overline{\lambda}$ are label-equivalent and fwd-equivalent.

($\Leftarrow$) Given solution $\overline{\lambda}$, we show that there exists a label-equivalence solution λ. Consider an arbitrary $\overline{v.i}$. Suppose $(A, \overline{v.i}, \overline{v.i'}) \in \overline{\mathrm{choice}}_{\lambda}(\overline{v})$. From the $\forall\exists$ abstraction, we know that there must be at least some $(v.i, v.i')$ such that $f(v.i, v.i') = (\overline{v.i}, \overline{v.i'})$. For an arbitrary such $(v.i, v.i')$, we know that $A = \overline{\eta}(f(v.i), \overline{\lambda}(f(v))))$. By IH, we have $\overline{\lambda}(f(v)) = h(\lambda(v))$. Therefore, $A = \overline{\eta}(f(v.i), h(\lambda(v))) = h\ (\eta(v.i,\ \lambda(v)))$. Let a stand for $\eta(v.i, \lambda(v))$. Then, $\eta(v.i, \lambda(v)) = a$ and $h(a) = A$. By the definition of $\mathrm{choice}_{\lambda}$, it follows that $\exists a.h(a) = A \wedge (a, v.i, v.i') \in \mathrm{choice}_{\lambda}(v)$. From Lemma 2, this implies that we also have label-equivalence and fwd-equivalence. □

Corollary 2. *Suppose we have well-formed* $\mathcal{N}_{\mathrm{d}}^{\mathrm{s}}$, $\overline{\mathcal{N}}_{\mathrm{d}}^{\mathrm{s}}$ *and* $\widehat{\mathcal{N}}_{\mathrm{d}}^{\mathrm{s}}$ *and* $\overline{\widehat{\mathcal{N}}}_{\mathrm{d}}^{\mathrm{s}}$ *with the effective abstractions* (f, h) *and* $(\hat{f}, \hat{h})$*, respectively. If* $\overline{\mathcal{N}}_{\mathrm{d}}^{\mathrm{s}}$ *equals* $\overline{\widehat{\mathcal{N}}}_{\mathrm{d}}^{\mathrm{s}}$*, then for any* λ*, label and fwd-equivalence* $\hat{\lambda}$ *exist and vice versa.*

Corollary 3. *Suppose we have well-formed* $\mathcal{N}_{d}^{s}$, $\overline{\mathcal{N}}_{\mathrm{d}}^{\mathrm{s}}$ *and* $\widehat{\mathcal{N}}_{\mathrm{d}}^{\mathrm{s}}$ *and* $\overline{\widehat{\mathcal{N}}}_{\mathrm{d}}^{\mathrm{s}}$ *and the effective abstractions* (f, h) *and* $(\hat{f}, \hat{h})$*, respectively. If* $\overline{S|\mathcal{N}}_{\mathrm{d}}^{\mathrm{s}}$ *equals* $\overline{\widehat{S|\mathcal{N}}}_{\mathrm{d}}^{\mathrm{s}}$*, for any* λ*, then label and fwd-equivalence* $\hat{\lambda}$ *exist and vice versa.*

The proofs of corollaries can be made by transitivity; details are provided in the appendix.

4.3. Computation

Through the corollaries presented above, we can check whether the network instance and its updated instance are equivalent. Algorithm 3 starts by checking whether the updated configuration snippets occur at the nodes in the network subinstance (lines 2–5). If the

Algorithm 3: Equivalence Function $Equal(n, n', c, c')$.

```
1  flag ← False                ▷ flag shows whether instance n and updated instance n' are
   equivalent. c and c' are the original and updated configuration;
2  (cc[.], lc[.]) ← COMPARE(c, c', n)   ▷ cc[.] and lc[.] are the updated configuration
   snippets and locations respectively;
3  if CONNECT(cc[.], lc[.], n, c) then
4      flag ← True
5      return flag
6  an ← GETABSTRACTION(n, c)            ▷ an is abstract instance for n;
7  an' ← GETABSTRACTION(n', c')
8  if ISSAME(an, an') then
9      flag ← True
10     return flag
11 else if SUBINSTANCE(an, an') then
12     flag ← True
13 return flag

   Function: GetAbstraction(n, c)
14 an ← INITIALIZE(n.d, c)
15 while VIOLATECONDITION(an, c) do
16     an ← REFINE(an, c)
17 return an
```

abstract instances with the original configuration c and updated configuration c' are the same, network instance n is equivalent to the updated instance n' (lines 6–10). If the network subinstances of these abstract instances are equal, their forwarding behaviors are equivalent (lines 11–13). We then show how to compute network subinstances and abstract network instances. To obtain a network subinstance, computing all the loop-free paths from destination to source is time-consuming. In fact, we only need to determine whether the paths from destination to source traverse the ports with updated configuration snippets. By analyzing the configurations, we can obtain the changed configuration snippets cc[.] and their locations lc[.] related to the destination prefix of the network instance (line 2). After removing the destination ports and their connected links in the network instance, if the source and updated nodes (with added route map, removed static route, etc.) are not in the same connected component, the subinstances are equivalent (lines 3–5). The key to obtaining an abstraction instance is to compute the appropriate topology function f and attribute function h. Unlike computing an abstract control plane instance [9], we take ACLs into account. We first fix the attribute function because it depend only on the protocol. To compute the topology function, we start by specializing the transfer and ACL functions. We map the concrete destination node to a single abstract

destination node and all other concrete nodes to other abstract nodes (line 14). When this abstraction violates some condition in the definitions (e.g., abstraction, acl-equivalence), we break up the problematic abstract node into multiple abstract nodes (lines 15–16).

Computing the connected component for subinstances yields $\mathcal{O}(v(v+e))$ complexity, where v and e are the number of nodes and links, respectively. In the worst cases, the abstract network instance is the same as the concrete instance, indicating that each node in the abstract instance will be refined after checking the conditions. The complexity of each round of checking is $\mathcal{O}(n^2)$, where n is the number of conditions in the control plane transfer and ACL functions. Thus, the overall check complexity is $\mathcal{O}(vn^2)$. Combining the above parts yields a total complexity of $\mathcal{O}(v(n^2+v+e))$.

4.4. Discussion

NUV can accurately support configuration updates to static routes, connected networks, interior gateway protocols (e.g., OSPF), BGP, redistribution of routes between routing protocols, ACLs, policy routing, and so on. Thus, NUV is both generally applicable and flexible, but it does have some limitations. Networks use numerous protocols, and NUV does not currently support some protocol features, such as those used for traffic engineering (e.g., MPLS TE [11]) or nondeterministic protocols (e.g., RSVP). Even lacking such support, our evaluations of benchmark networks show that our models in NUV are sufficiently rich to capture the behaviors of many real, large networks. Configuration tools such as ERA [6], ARC [5] and Minesweeper [1]and Tiramisu [12] use the Batfish configuration parser, which does not support the MPLS TE or RSVP protocols, and these configuration verification tools do not model these protocols. In fact, to include additional protocols and features, the configuration parser could gradually be improved by implementing appropriate translation routines; in theory, NUV is able to support these protocols by adding some extensions. In addition, any improvements made to configuration verification work will benefit NUV because the inference and reduction in the framework is based on a network instance that models the effect of configuration on routing and forwarding.

Another question is: Can the output of NUV cover all impacted queries caused by all the updates in the configurations? The workflow of NUV includes two key stages. First, NUV uses Algorithm 2 in Section 3.2 to infer the potentially impacted traffic (denoted as set B) by exploring the effect of changed configuration in the network model. Next, using a variety of conditions that are designed to be checked locally, NUV validates the equivalence of the forwarding behavior in the original and updated configuration for each potential impacted traffic; then, it removes the traffic (denoted as set C), each of which has the equivalent behavior under the original and updated configuration, from the potential impacted traffic (set B). In the first stage, for the network instance of each traffic, if the output of a transfer function, ACL function or selection function produces updates under the changed configuration, the traffic is added to the impacted potential impacted traffic (set B). For each traffic in a real traffic set impacted by the configuration update, a transfer function, ACL function or selection function must exist whose output will be altered under the changed configuration. Using reductio ad absurdum logic, we know that if the outputs of the transfer function, ACL function or selection function remain unchanged, the data forwarding solution of the traffic will not be impacted by the configuration update. Therefore, all impacted traffic is added to set B. In the second stage, each traffic in set C has behavior equivalence under the original and updated configurations, as we proved in Section 4. In other words, the traffic in set C is not impacted by the configuration update. Therefore, all the impacted traffic lies in sets (B-C).

5. Implementation and evaluation

We implemented the framework described above in Java. Given original and updated configurations, in the first stage, we aim to obtain the detailed updated configuration snippets. The languages used for input configuration are defined by multiple vendors, such as Juniper, Arista, and Cisco. Thus, we first need to obtain vendor-agnostic configurations and then operate over these representations. Today, most of the state-of-the-art configuration verification tools use the Batfish configuration parser [4] to convert a vendor-specific configuration into a vendor-agnostic format. We also leverage the Batfish [4] parser and implement the inference task based on the algorithms in Section III. Using the algorithm in Section IV, we implement the reduction task, and finally, we obtain the verification queries for the impacted traffic. The output of our tool—the impacted queries—can act as guidance and input for a series of configuration verification tools (e.g., ARC [5], ERA [6], and Minesweeper [1]). Note that NUV is open source [13].

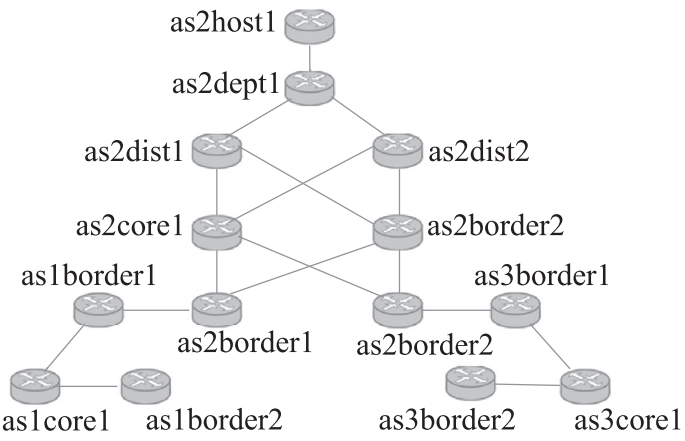


Fig. 7. A campus network update example [14].

Next, to evaluate NUV, we must answer the following questions. (i) How efficiently can NUV generate impacted queries for configuration updates? (ii) How much does it improve the speed of the configuration verification task? To answer these questions, we applied NUV to benchmark networks, including both synthetic and real networks [14].

All the experiments were executed on a server running the Ubuntu 10.12 operating system and equipped with an 8-core, 2.4-GHz Intel i7 processor.

In this section, we first demonstrate NUV's ability to determine the queries for the impacted traffic based on the use cases. Then, we test the effectiveness of NUV through a set of performance evaluation experiments.

5.1. Case studies

Using the benchmark configurations named CAMPUS1 and CAMPUS2 [14], we observe that there was an update to the router configurations. We leverage NUV to determine the impacted verification queries. As Fig. 7 shows, the network includes 14 routers running OSPF within the same AS and running BGP for interdomain connectivity. According to their descriptions, under their original configuration, prefix hijacking was active in the network. Specifically, an eBGP advertisement from AS3 (with a prefix of as2dept1 2.128.0.0/9) will override the BGP decision process and cause the as2border2 router to forward traffic whose destination is as2dept1 to AS3 instead of to the router as2dept1. The updated configuration performs a configuration update on the router as2border2 that interconnects the main campus with the satellite campus AS3. The detailed updated route map is as follows.

```
neighbor as3 route-map as3_to_as2 in
ip prefix-list outbound_routes seq 5 permit 2.128.0.0/9 ge 16 le 32
route-map as3_to_as2 deny 90
 match ip address prefix-list outbound_routes
```

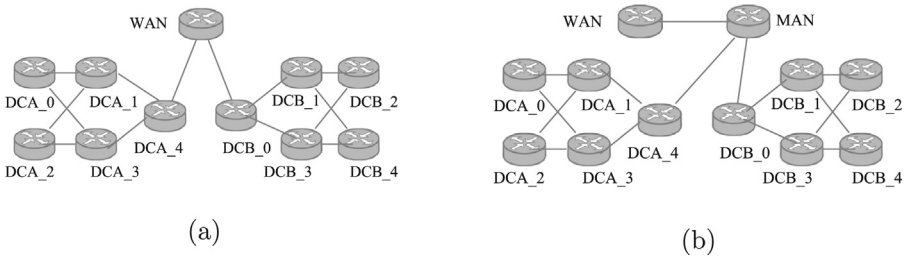


Fig. 8. A network update example from NetCraft [3].

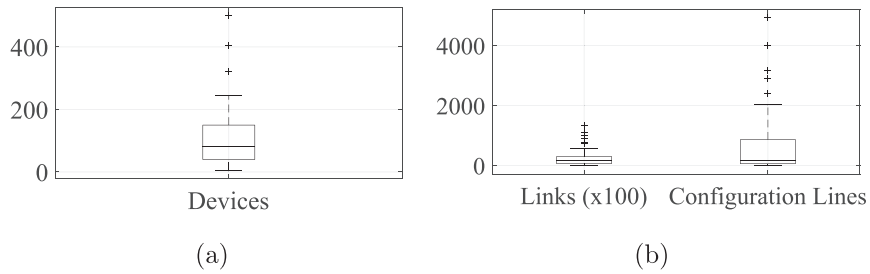


Fig. 9. The scales of the analyzed networks.

We then applied NUV to investigate and analyze the original and updated configurations. NUV's output consists of 40 queries related to traffic whose destination is 2.128.0.0/9. The configuration snippet updates are of type II from Table 2. According to NUV's inference, only a few network instances with a specific destination will be impacted. Without NUV, 1560 (40 × 39) queries would need to be verified under the updated configuration.

Fig. 8 shows an example of a network update from NetCraft [3]. In this example, the enterprise network structure has underg1 a major adjustment. In the original network, as shown in Fig. 8(a), DCA and DCB directly peer with the WAN. In the updated network in Fig. 8(b), DCA and DCB peer with the MAN, and only MAN peers peer with the WAN. We construct the configurations according to the design where the border routers between DC, WAN and MAN run eBGP for interdomain connectivity, and each DC has five routers running OSPF for intradomain connectivity. Without NUV, approximately 272 verification queries must be verified. After using NUV to determine the impacted verification queries, only 150 queries require verification. According to NUV's inference, the updates are of type III, and all the network instances except those with intradomain sources and destinations will be impacted. For this case, we then analyze the output queries and find that the intradomain traffic queries in each data center are out of scope. This conclusion is reasonable because the updated eBGP neighbors will not impact the intradomain traffic. The evaluation shows that even when a network undergoes major adjustments, NUV can help reduce the number of required verification queries.

5.2. Benchmark networks

For the performance evaluation, we use the configuration files of 42 available benchmark networks [14]. These network configurations are functional and include OSPF, iBGP, eBGP, static routes, ACLs and so on. These networks include from 2 to 500 routers with between 4 and 4950 total configuration lines and 0.6–133k links. The box plot in Fig. 9 shows the network sizes, including the number of devices, the number of links and the number of configuration lines, and reflects the distribution of each number, including the minimum, first quartile, median, third quartile, and maximum. In addition to these network topology factors, we also provide the number of traffic equivalence classes in these networks. As introduced earlier, the equivalence classes in a network determine the size of all-pair queries to some extent. We analyze the number of traffic equivalence classes of these networks. The result is visualized by the cumulative distribution function (CDF). The cumulative distribution function of a random variable is a method for describing the distributions of random variables. For the CDF, F (x) accumulates all the probability less than or equal to the independent variable x. As shown in Fig. 10(a), there are fewer than 6560 traffic equivalence classes in 90% of the networks and fewer than 3430 equivalence classes in 80% of the networks. Fig. 10(b) shows the number of traffic equivalence classes for these networks; the networks with more traffic equivalence classes appear further to the right.

To evaluate NUV's performance, we need to input the original and updated configurations into our tool. In addition to the several available updated configuration files, we also generate updated configurations for these networks to perform more performance evaluations. As the impacted traffic is determined by the updated configuration snippets, we obtain the normally functioning updated configurations for each network by carefully considering the following factors. (i) Type: The updated configuration snippets cover all update types in Table 3. (ii) Distribution: The configuration updates apply to more than a quarter of the devices in each network. (iii) Quantity: The number of changed configuration snippets (e.g., ACL static route, BGP peer) is more than a quarter of the number of devices.[4] The configurations described above are freely available [13].

5.3. NUV Efficiency

NUV outputs the impacted queries to be verified due to configuration updates. We now evaluate the efficiency of NUV on determining the impacted queries to be verified. We measure the total computation time for the impacted queries of NUV with each configuration and its corresponding updated configuration. We analyze each network five times with NUV to obtain an average execution time. Fig. 11(a) shows the average execution times for these networks. The details of these benchmark networks were described in the previous subsection. In these networks, the number of traffic equivalence classes increases as the numbers of nodes, links and policies increase. The evaluated networks are sorted by the number of traffic equivalence classes; the more complex networks appear further to the right in Fig. 11(a). Similar to other verification works, e.g., [1] and [5] [4], it is almost infeasible to define the

[4] In real-world situations, users do not usually update so many devices. The cases that we evaluate here are more complex to ensure that the performance of NUV is well evaluated.

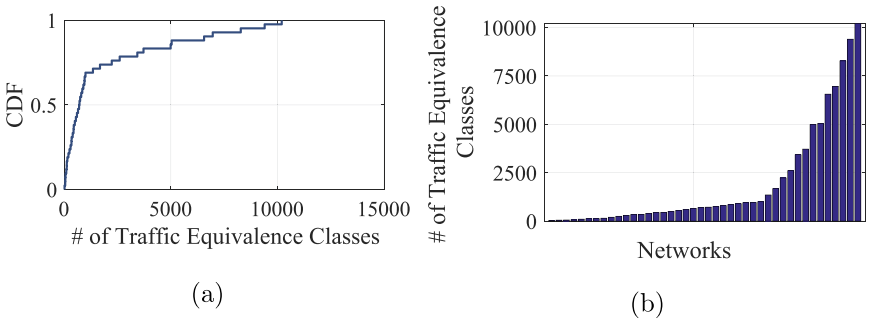


Fig. 10. The numbers of traffic equivalence classes.

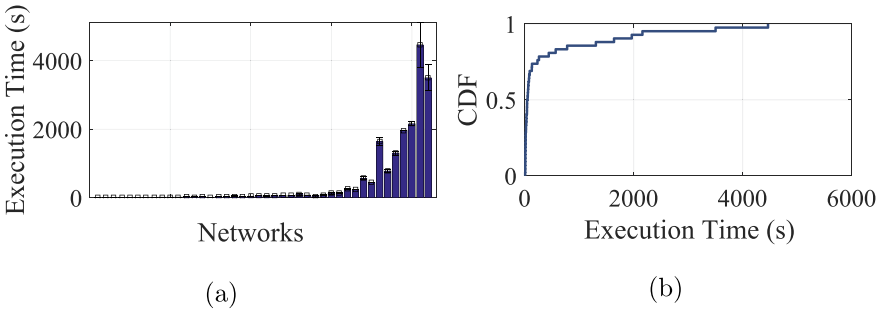


Fig. 11. Time required to determine queries in the network.

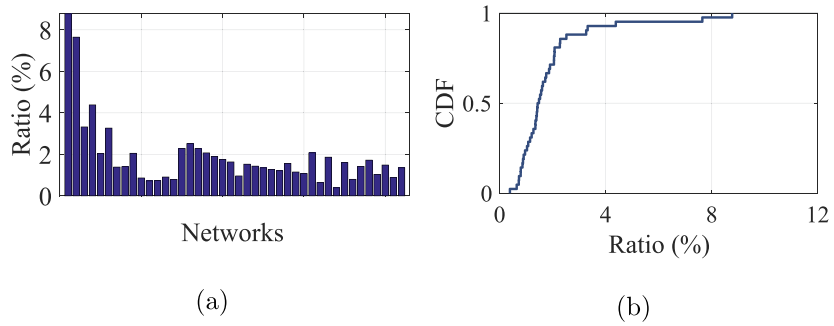


Fig. 12. Ratios of impacted queries to all queries.

performance of the tool as a detailed function of the numbers of nodes, links, or policies. Thus, we do not focus on analyzing the relationships between the execution time and the numbers of nodes, links, or policies. For the benchmark networks [14], the nodes, links and policies grow linearly so that the complexities of the network configurations are similar. NUV execution requires an average of 441.16 seconds across all the studied networks. The average execution time for each network ranges between 6.56 seconds and 4,463.10 seconds. From Fig. 11(b), we can see that the execution times are below 790 seconds for 85% of the networks and below 480 seconds for 80% of the networks. The above execution times include both the configuration parsing and computation times. After further analysis, we found that for networks with fewer equivalence classes, the majority of the time is spent on parsing the network configurations.

As mentioned above, NUV outputs the impacted queries for updated networks. Next, we examine the number of output queries for these networks. Fig. 12(a) shows the ratio of impacted traffic for the tested networks. The networks are sorted by number of traffic equivalence classes; the more complex networks appearing further to the right. The number of impacted queries ranges from approximately 30 to 1,407,013. In contrast, the total number of queries for these networks ranges from 342 to 104,050,200. We analyze the ratio of impacted queries to all queries for each network and show the results in Fig. 12(a). We observe that the ratio ranges from 0.3387% to 8.7719%. As shown in Fig. 12(b), after further analysis, in 90% of the networks, this ratio is below 3.26%, and for 80% of the networks, it is less than 2.07%. Although the ratio seems substantial, we are more concerned with NUV's verification task speed improvements. In the next subsection, we compare the speed of NUV-based configuration verification against that of configuration verification without NUV.

5.4. Verification efficiency

As introduced earlier, the output of NUV consists of guidance for executing configuration verification tools. We measure the efficiency of NUV's guidance on configuration verification using Minesweeper [1], a state-of-the-art network configuration verification tool and compare the overall execution time with NUV to the overall time without NUV. Specifically, the former includes both the time to determine the impacted queries and the time to verify those queries using Minesweeper, while the latter refers to the time needed to verify all-pair queries using Minesweeper. Arguably, it might be inappropriate to compare the overall time required when using NUV to the verification time required for all-pair queries because operators may rule out some queries using heuristic strategies based on experience. However, on one hand, the operators cannot guarantee that the verification results of removed queries would indeed remain unchanged and thus do not require verification. On the other hand, even with these heuristics strategies such as management interface reachability, very large numbers of queries must still be verified [1]. Thus, a comparison with the verification time required for all-pair queries could be a helpful basic reference.

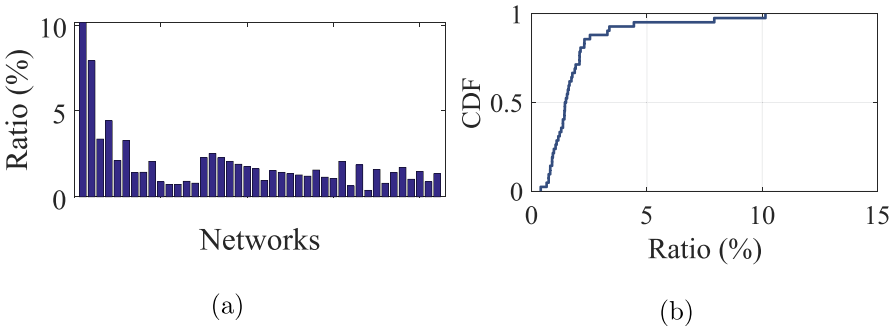


Fig. 13. The ratio of the total execution time with NUV to the time without NUV.

Note that for most of the benchmark networks, the number of queries that must be verified without NUV is far greater than 40,000; consequently, even if a single query requires only 10 seconds, full verification without NUV will take at least a couple of days. Similar to other works, such as ARC [5], to address the problem of large number queries to be verified, we use a function to obtain the total verification time for this performance evaluation comparison. This is a multiplication function that multiples (1) the number of queries with (2) the average time needed for single-query verification. In our evaluation, we use Minesweeper to verify all the queries. For networks with each type of topology, we start the evaluation with the smallest network on our server; the more complex networks are verified later. When the verification time exceeds one day but the verification task is not completed, we will stop the verification, and the multiplication function is used to obtain the total verification time. The average time for single-query verification used in the multiplication function is obtained through experimental verification for these benchmark networks. Specifically, for each network, ten queries are randomly selected and evaluated with Minesweeper on our server, yielding the average time for single-query verification can be obtained. Based on the interim and final output of NUV, we can obtain the number of impacted queries and all queries; then, the verification time can be obtained using the multiplication function.

With NUV, the overall execution time for these networks ranges from approximately 46.69 seconds to 1.9671×10^{10} seconds. In contrast, the total verification times without NUV range from 460.39 seconds to 2.2293×10^{12} seconds. To compare the above execution times, we further computed the ratio of the total time with NUV to that without NUV. Fig. 13(a) presents the ratio values of these networks, showing that the maximum value of the ratio is less than 10.1423%. Fig. 13(b) shows the cumulative distribution function of the ratio. We can see that for 90% of the networks, the ratio is below 3.37% and for 80% of the networks, the ratio is less than 2.29%. There is an argument that—even with NUV—the overall execution time may exceed 1.9×10^{10} seconds. The truth is that a single query for a large network in our experiment takes more than 5000 seconds with the state-of-the-art Minesweeper tool. However, NUV reduces the overall execution time by two orders of magnitude, making the total verification task for large networks more feasible.

Table 4 shows the comparison result for configuration verification time with or without NUV. For the configuration verification time with NUV, we take both the determination of the queries to be verified and the verification of these impacted queries into account. All times are in seconds. The studied networks consist of real networks and synthetic networks from Batfish. We sort these networks by the types of network topologies: Fat-tree, Ring, Fullmesh and others. For networks of each type of topology, we provide the execution time for the smallest and the largest networks and show their nodes, links and number of configuration lines. Note that the verification time for the smallest network with each type of topology in Table 4 is obtained through experimental verification, while the verification time for the largest networks are obtained by the multiplication function. As shown in Table 4, when using NUV for verification, the time spent on query determination is far less than that on the verification for impacted queries. In the worst case, the impacted query determination with NUV takes 4,463.10 seconds. In that case, query determination time is much less than the verification time for a single query in that network (i.e., the largest Fullmesh network). The query determination time in the largest networks with other topologies is also less than the single-query verification time. In addition, we observe that, with NUV, for the largest Fullmesh network and the largest Fat-tree network, the verification time exceeds a few days. This time is mainly due to the poor scalability of the single-query verification in these networks. For example, verifying a single query in the largest Fullmesh network takes approximately one and a half days. The verification with NUV for the networks with ring topology and others can be done in less than a day. The largest ring network takes 48,347.91 seconds (approximately 14 hours), to complete the verification task; in contrast, without NUV, it takes more than 100 hours. The maximum verification time with NUV for networks with other topologies is 980.20 seconds; in contrast, verification without NUV takes 20,730.40 seconds. Thus, NUV significantly accelerates the configuration verification task. Specifically, the ratio of the overall execution time with NUV to the execution time without NUV can show how much NUV improves the speed of the configuration verification. For networks with each type of topology, Table 4 shows the highest and lowest ratios found during the evaluation. The ratio for the Fullmesh networks ranges from 0.39% to 10.14%. The ratios for ring networks range from 0.73% to 7.92%. The highest and lowest ratios for Fat-tree networks are 3.28% and 1.35%, respectively. In addition, we can see that the average ratio for Fat-tree, Ring, and Full-mesh networks is lower than that of the other network types. Through further analysis, we found that the Fat-tree, Ring, and Full-mesh networks have an almost symmetric design. In other words, the ratio for networks with symmetric characteristics is much lower. This is reasonable; based on the previous theoretical introduction, we know that these networks are more likely to satisfy the local conditions when checking whether an updated network instance is equivalent to the original network instance.

Next, with Fig. 14, we can see that, in the networks of each type of topology, the highest ratio (the total time with NUV to the time without NUV) usually corresponds to the smallest network.[5] In addition, as shown in Fig. 13(a), the ratio of a small network is usually higher than that of a large network. After sorting by the number of traffic equivalence classes, the top 50% belong to large networks, and the other half of belong to smaller ones. Then, we observed that the ratios of the latter average 1.56 times larger than those of the former. This is reasonable because the ratio is formed by dividing the time with NUV (which is the sum of the determination and verification times for impacted queries) by the time without NUV, and when using state-of-the-art verification tools, such as Minesweeper, the verification time for a single query grows ex-

[5] There are only a few networks with other topologies; thus, we provide the ratio for Fat-tree, Ring and Fullmesh networks.

Table 4
Statistics of comparison result.

Topology	Size	Nodes / Edges	Time with NUV (s)		Time without	Ratio		
		Configuration lines	Determination	Verification	NUV (s)	Highest	Lowest	Average
Fullmesh	Smallest	10 / 45 / 2000	6.56	40.13	460.39	10.14	0.39	1.81
	Largest	100 / 4950 / 110,000	4,463.10	1.9671×10^{10}	2.2293×10^{12}			
Ring	Smallest	10 /10 / 1300	8.25	222.39	2,911.80	7.92	0.73	1.95
	Largest	200 / 200 / 26,000	89.73	48,347.91	4.1279×10^{6}			
Fat-tree	Smallest	5 / 4 / 616	9.37	1,663.90	50,971.90	3.28	1.35	1.87
	Largest	500 / 4000 / 133,600	3,500.90	1.9626×10^{8}	1.4514×10^{10}			
Others	Smallest	26 / 33 / 1183	8.15	562.06	16,929.23	4.43	3.37	3.90
	Largest	26 / 33 / 1,186	10.52	980.20	20,730.40			
All types	–	–	–	–	–	10.14	0.39	1.94

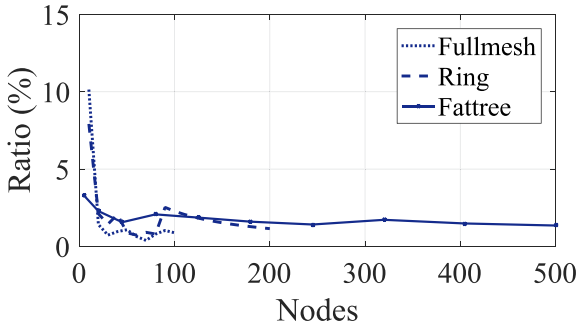


Fig. 14. The ratio of the execution time with NUV to the time without NUV for networks with different topologies.

ponentially as the network size increases [9]. In some cases, query determination time is much less than the verification time for a single query in large networks. In contrast, for a very small network, the time to complete verification without NUV is small; thus, the ratios are comparatively higher, up to almost 10%. In practice, verifying a query in a large network with hundred devices takes approximately half an hour [1,9]. Thus, with NUV, configuration verification can be significantly accelerated.

6. Related works

Data Plane Verification: Recent efforts in network verification [7,15–19] have also focused on network data planes. SymNet [19] applies symbolic execution to analyze an abstract data plane model written in SEFL language; it analyzes every node and the links between nodes. Compared with other data plane verification tools (e.g., Anteater [15], HSA [17], NOD [7], VMN [18]), SymNet has the most comprehensive coverage for analyzing widely deployed network functionalities. Another work, SFC-Checker [20], was proposed to check the correctness of the stateful forwarding behaviors of SFC chaining in a network. SFC-Checker uses an abstract stateful data plane model that captures the high-level states maintained by a network function (NF). Compared to Symnet, SFC-checker can model different packet sequences and their resulting NF states. In addition, VeriGraph [21] models a network and the involved stateful network function as a set of first-order logic (FOL) formulas; those formulas can be passed to the SMT solver Z3 to verify some reachability-based properties on service graphs. VeriGraph enriches more NF model catalogs and is more general and can easily be applied to other network function virtualization scenarios. SymNet, SFC-checker and VeriGraph do not focus on proposing techniques for verifying network properties to deal with typical updates on the data plane. In fact, only small parts of the network tend to be affected by the updates at the data plane level; therefore, several frameworks, such as VeriFlow [22], NetPlumer [23] and Delta-net [24], have proposed performing real-time validation of network rule updates instead of validating all rules on the data plane. However, almost all these verification tools analyze the network's current data plane, which prevents them from being used for proactive analysis to avoid undesirable forwarding behaviors. Unlike these tools, our tool focuses on speeding up control plane verification when network configuration files are updated. In short, the existing verification works, such as VeriFlow, NetPlumer and Delta-net, exploit a similar intuition that provides an optimization approach for network verification during network updates; however, those tools operate over the network data plane (i.e., the packet-forwarding rules), whereas NUV operates over the control plane (i.e., the protocols that distribute the available routes). Compared the relationship between the rules on data plane and packet forwarding, the relationship between configuration snippets and forwarding behavior is much more complex. The processing of a routing message depends on a wide range of configuration snippets, and the routing messages can interact with one another; thus, the verification for the configuration updates is much more challenging than the verification of the rule updates on data plane. Moreover, unlike NUV, the existing verification tools cannot determine the impacted queries proactively.

Control Plane Verification and Compression: Data plane verification tools cannot prevent errors proactively—before undesirable forwarding occurs. To address this issue, several control plane verification frameworks have been developed to detect potential configuration errors proactively. For instance, Batfish [4] takes the network configuration files and the current network environment (i.e., route advertisements) as input. It simulates the behavior of routing protocols to generate the network data plane and then verifies the network properties based on the resulting data plane. However, this simulation becomes excessively expensive when applied to large networks under link failures. To address the challenge of verifying networks under arbitrary failures, Aaron et al proposed ARC [5], which models the network control plane as abstract weighted graphs; then, the network properties can be verified using polynomial-time graph algorithms. However, the network model in ARC cannot capture the forwarding behavior of networks with certain routing features (e.g., BGP local preference feature). Tiramisu [12] introduced an improvement on ARC by embellishing the graph model with richer graph constructs and then using a combination of graph traversal algorithms and Integer Linear Programs (ILPs) to verify network properties. But Tiramisu cannot be applied to verify control plane equivalence, because it cannot symbolically model advertisements. Another tool, ERA [6], uses binary decision diagrams (BDDs) to represent a concrete set of control plane messages. Based on the current router configurations, each router is translated into a transfer function of this set; then, ERA propagates this set along a path through the network. Like Tiramisu, ERA cannot be applied to verify control plane equivalence. Minesweeper [1] uses an abstract record of symbolic values to represent the protocol messages. It uses a logical formula to capture the forwarding behavior for the networks and then combines the formula with constraints that describe the intended network property. Using an SMT solver, Minesweeper checks whether the intended network property holds in the face of all adver-

tisements. But its verification performance is very poor, despite many internal SMT-specific optimizations. Plankton [25] used symbolic partitioning to manage large header spaces and adopted model checking to explore protocol behavior. Plankton works efficiently on data center networks with symmetry. However, because of the inherent limitations of model checking, Plankton is challenged for scaling to large networks. In short, for a single query in a large network, almost all the existing control plane verification tools can take hours, if not days, to verify complex scenarios. And as pointed in Bonsai [9], the costs of these verification tools can grow exponentially in the worst case. Our work, NUV, is complementary to these efforts. NUV can capture the queries for impacted traffic with updated configurations. By preprocessing upcoming configuration changes, the purpose of NUV is to reduce the number of queries that must be executed upon configuration changes. Bonsai [9] proposed a network control plane compression algorithm that preserves reachability properties for traditional networks; however, Bonsai is intended to speed up the analysis for a single query, whereas we focus on speeding up the total verification task by determining the impacted queries. Like existing configuration verification tools (e.g., Minesweeper, Batfish, ERA), NUV focuses on networks running traditional distributed network protocols. NUV cannot verify network properties in the service graph in Network Function Virtualization (NFV), but it can help configuration verification tools to check network properties in traditional networks.

Network Configuration Synthesis: A rich literature on network configuration synthesis exists. Propane [26] allows users to specify their policies effectively and then automatically translates the specifications into router-level BGP configurations using an effective intermediate representation. Propane/AT [27] improves Propane by adding new abstractions, and it can execute at up to two orders of magnitude faster than Propane. Both of Propane and Propane/AT produce BGP configurations from high-level policies but neither supports multiple routing protocols. Another tool, SyNET [28], reduces the problem of finding correct input configurations to the task of synthesizing inputs for a stratified Datalog program. SyNET supports multiple interacting routing protocols but does not scale to realistic networks. Unlike the existing approaches which directly generate policy-compliant configurations, Zeppelin [29] uses policy-compliant paths to guide the synthesis process and can generate policy-compliant control planes that also behave well under the majority of small network failures. Another tool, NetComplete [30], is more easily used; it allows users to precisely control how configurations should be changed. It takes incomplete configurations in which some parameters remain to be completed, automatically finds these incomplete parameters, and then completes the configuration by adding concrete values that reflect the users' policies. These tools require formal specifications of the network policies as input and output network configurations. Our work enables users to check traffic impacted by such generated configurations, which can help to improve their designs.

Network Configuration Update: Many situations, such as incident mitigation or new application requirements, require frequent network configuration updates. However, manually updating configurations is always both time-consuming and error-prone. NetCraft [3] allows users to create fine-grained configurations with the flexibility to deactivate or undo any configuration change to enable safer configuration updates. However, NetCraft is still in a too-early stage to be used in practical networks. Other works, such as Dionysus [31], use graphs to model the dependencies among updates at the switches and then dynamically schedule these updates to perform smooth transitions. Using a novel network model, zUpdate [32] is able to perform congestion-free network updates under traffic matrix changes. Both of these tools focus on providing smooth transitions to data plane rule changes for software-defined networks. Recently, JinJing [33] proposed a method for updating the ACL configurations in Alibaba's global WAN, but it does not support configuration updates related to routing protocols. Our work aims to determine the impacted queries for updated configurations; thus, it is orthogonal to these works.

7. Conclusion

In this paper, we have argued that users usually lack the knowledge to precisely determine input queries and that verifying all-pair queries is intractable for network configuration verification. We have shown how more precise configuration verification can be accomplished by determining the queries impacted by updated configurations. We presented a systematic technique called NUV that infers and reduces the impacted queries using a network forwarding model. The outputs of NUV can be applied to a rich collection of existing network configuration verification tools. Evaluations on a series of benchmark networks showed that NUV determines impacted queries efficiently and can speed up the configuration verification process by several orders of magnitude.

Declaration of Competing Interest

The authors declare that we do not have any commercial or associative interest that represents a conflict of interest in connection with the work submitted.

CRediT authorship contribution statement

Yahui Li: Conceptualization, Methodology, Software, Investigation, Writing - original draft. **Zhiliang Wang:** Supervision, Conceptualization, Methodology, Formal analysis, Validation, Resources, Writing - review & editing. **Xia Yin:** Resources, Visualization, Writing - review & editing. **Xingang Shi:** Resources, Visualization, Writing - review & editing. **Jianping Wu:** Resources, Writing - review & editing. **Fangdan Ye:** Data curation, Validation, Visualization, Writing - review & editing. **Jiangyuan Yao:** Validation, Visualization, Writing - review & editing. **Han Zhang:** Visualization, Writing - review & editing.

Acknowledgements

We thank the anonymous reviewers and editor for their comments. This work was supported in part byProgram of Hainan Association for Science and Technology Plans to Youth R&D Innovation (QCXM201910), the National Key Research and Development Program of China under Grant 2018YFB1800205, the National Natural Science Foundation of China (61802092) and Scientific Research Setup Fund of Hainan University (KYQD (ZR) 1837).

Appendix A

Corollary 1. *Suppose we have a well-formed* $\mathcal{N}_{\mathrm{d}}^{\mathrm{s}}$ *and that* $\mathcal{N}'^{\mathrm{s}}_{\mathrm{d}}$ *is obtained by pruning* $\mathcal{N}_{\mathrm{d}}^{\mathrm{s}}$ *(i.e., removing some of the ports that are not in* $\mathcal{S}|\mathcal{N}_{\mathrm{d}}^{\mathrm{s}}$*). For any solution* λ' *of* $\mathcal{N}'^{\mathrm{s}}_{\mathrm{d}}$*, an equivalent* λ *of* $\mathcal{N}_{\mathrm{d}}^{\mathrm{s}}$ *exists.*

Proof. For a network instance $\mathcal{N}_{\mathrm{d}}^{\mathrm{s}}$ and its subinstance $\mathcal{S}|\mathcal{N}_{\mathrm{d}}^{\mathrm{s}}$, through Theorem 1, we know that solutions λ and $\overline{\lambda}$ are label-equivalent. We obtain that $\forall v.\ \lambda(v) = \overline{\lambda}(v)$. For a network instance $\mathcal{N}'^{\mathrm{s}}_{\mathrm{d}}$ and its subinstance $\mathcal{S}|\mathcal{N}'^{\mathrm{s}}_{\mathrm{d}}$, we know that the solutions λ' and $\overline{\lambda'}$ exist and are label-equivalent such that $\forall v.\ \lambda'(v) = \widehat{\lambda'}(v)$. The subinstance of $\mathcal{N}'^{\mathrm{s}}_{\mathrm{d}}$ is also $\mathcal{S}|\mathcal{N}_{\mathrm{d}}^{\mathrm{s}}$ because it is obtained by removing ports from $\mathcal{N}_{\mathrm{d}}^{\mathrm{s}}$ that are not involved in$\mathcal{S}|\mathcal{N}_{\mathrm{d}}^{\mathrm{s}}$. Thus, the subinstances of $\mathcal{N}_{\mathrm{d}}^{\mathrm{s}}$ and $\mathcal{N}'^{\mathrm{s}}_{\mathrm{d}}$ are the same, $\forall v.\ \overline{\lambda}(v) = \widehat{\lambda'}(v)$. By transitivity, we have $\forall v.\ \lambda(v) = \widehat{\lambda'}(v)$. □

Lemma 1. *Suppose we have the well-formed network instance* $\mathcal{N}_{\mathrm{d}}^{\mathrm{s}}$, $\overline{\mathcal{N}_{\mathrm{d}}^{\mathrm{s}}}$ *and an effective abstraction for data plane choice-equivalence; then, the abstraction has label-equivalence.*

Proof. Looking at the definition of λ, there are two cases to consider. First, we observe that when $\mathrm{node}(v.i) = s$, $\lambda(s) = a_{\mathrm{s}}$. It follows that $\overline{\lambda}(f(s)) = \overline{\lambda}(\overline{s}) = \overline{a_s} = h(a_{\mathrm{s}})) = h(\lambda(s))$. In the second case, we show the implications separately.

Case (⇒) Assume $\lambda(v) = a$. By the definition of λ, we know that there are some links such that $(a, v.i, v.i') \in \text{choices}_\lambda(v)$. Consider all the concrete links $(a, v.i, v.i') \in \text{choices}_\lambda(v)$. From choice-equivalence, we know that $(h(a), f(v.i), f(v.i')) \in \text{choices}_\lambda(f(v))$ for each such pair. By the definition of λ, we then know that $\overline{\lambda}(f(v)) = A = h(a)$. By transitivity, $\forall v.i.\ h(\lambda(v)) = \overline{\lambda}(f(v))$.

Case (⇐) Assume $\overline{\lambda}(f(v)) = A$. We know that $A \in \text{choices}_\lambda(f(v))$. Assume $(A, \overline{v.i}, \overline{v.i'}) \in \text{choices}_\lambda(f(v))$. Consider all such $(A, \overline{v.i}, \overline{v.i'}) \in \text{choices}_\lambda(f(v))$.

From choice-equivalence, we know that for any concrete $v.i$ that there exists an $f(v.i) = \overline{v.i}$ such that $h(a) = A$ and $(a, v.i, v.i') \in \text{choices}_\lambda(v)$. Therefore, $a \in \text{choices}_\lambda(v)$. Finally, we obtain that the labeling can be $(\lambda(v) = a)$. It follows from transitivity that $h(\lambda(v)) = \overline{\lambda}(f(v))$. □

Lemma 2. *Suppose we have the well-formed network instances $\mathcal{N}_d^s$ and $\overline{\mathcal{N}_d^s}$ and an effective abstraction for data plane choice-equivalence; then, the abstraction has fwd-equivalence.*

Case (⇒) We assume $(v.i, v.i') \in \text{fwd}_\lambda(v)$ and need to show that $(f(v.i), f(v.i')) \in \overline{\text{fwd}}_{\overline{\lambda}}(f(v))$. By the definition of fwd_λ, we know that $\exists a.(a, v.i, v.i') \in \text{choices}_\lambda(v)$. From choice-equivalence, this means that $(h(a), f(v.i), f(v.i')) \in \overline{\text{choices}}_\lambda(f(v))$. Because we have label equivalence, $h(\lambda(v)) = \overline{\lambda}(f(v))$. By the definition of $\overline{\text{fwd}}_{\overline{\lambda}}$, $(f(v.i), f(v.i')) \in \overline{\text{fwd}}_{\overline{\lambda}}(f(v))$.

Case (⇐) We assume $(\overline{v.i}, \overline{v.i'}) \in \text{fwd}_\lambda(\overline{v})$ and show that a $v.i'$ exists such that $(v.i, v.') \in \text{fwd}_\lambda(v)$. By the definition of fwd_λ, we know that $\exists A.(A, \overline{v.i}, \overline{v.'}) \in \overline{\text{choices}}_\lambda(\overline{v})$. From choice-equivalence, this means that $\forall \overline{v.i} = f(v.i), \exists a.h(a) = A \wedge (a, v.i, v.i') \in \text{choices}_\lambda(v)$. Consider any such $(v.i, v.i')$ where $f(v.i, v.i') = (\overline{v.i}, \overline{v.i'})$. By rewriting slightly, we obtain $\exists a.(a, v.i, v.i') \in \text{choices}_\lambda(v) \wedge \overline{\lambda}(f(v)) \approx h(a)$. From transfer equivalence, we know that $\overline{\lambda}(f(v)) = h(\lambda(v))$, thus, $h(a) \approx h(\lambda(v))$, and therefore, $a \approx \lambda(v)$. Finally, from the definition of fwd_λ, we have $(v.i, v.i') \in \text{fwd}_\lambda(v)$.

Corollary 2. *Suppose we have well-formed $\mathcal{N}_d^s$, $\overline{\mathcal{N}_d^s}$ and $\widehat{\mathcal{N}_d^s}$ and $\overline{\widehat{\mathcal{N}}_d^s}$ with the effective abstractions (f, h) and $(\hat{f}, \hat{h})$, respectively. If $\overline{\mathcal{N}_d^s}$ equals $\widehat{\overline{\mathcal{N}}_d^s}$, for any λ, label and fwd-equivalence $\hat{\lambda}$ must exist, and vice versa.*

Proof. (*i*) Suppose λ is a solution for $\mathcal{N}_d^s$. From Theorem 2, we know that a solution $\overline{\lambda}$ for $\overline{\mathcal{N}_d^s}$ exists, where λ and $\overline{\lambda}$ are fwd-equivalent. Looking at the definition of label and fwd-equivalence, we have $\forall v.\ h(\lambda(v)) = \overline{\lambda}(f(v))$ and $\forall v.i, v'.i'.\ (v.i, v'.i') \in \text{fwd}_\lambda(v) \Leftrightarrow (\overline{v.i}, \overline{v'.i'}) \in \overline{\text{fwd}}_{\overline{\lambda}}(\overline{v})$. Similarly, for $\widehat{\mathcal{N}}_d^s$ and $\overline{\widehat{\mathcal{N}}}_d^s$, we have $\forall \hat{v}.\ \hat{h}(\hat{\lambda}(\hat{v})) = \overline{\hat{\lambda}}(\hat{f}(\hat{v}))$ and $\forall \widehat{v.i}, \widehat{v'.i'}.\ (\widehat{v.i}, \widehat{v'.i'}) \in \text{fwd}_{\hat{\lambda}}(\hat{v}) \Leftrightarrow (\overline{\widehat{v.i}}, \overline{\widehat{v'.i'}}) \in \overline{\widehat{\text{fwd}}}_{\overline{\hat{\lambda}}}(\overline{\hat{v}})$. (*ii*) As $\overline{\mathcal{N}_d^s}$ equals $\overline{\widehat{\mathcal{N}}}_d^s$, we know that $\forall v.\ \overline{\lambda}(f(v)) = \overline{\hat{\lambda}}(\hat{f}(\hat{v}))$ and $\forall v.i, v'.i'.\ (\overline{v.i}, \overline{v'.i'}) \in \overline{\text{fwd}}_{\overline{\lambda}}(\overline{v}) \Leftrightarrow (\overline{\widehat{v.i}}, \overline{\widehat{v'.i'}}) \in \overline{\widehat{\text{fwd}}}_{\overline{\hat{\lambda}}}(\overline{\hat{v}})$. By transitivity, we have label-equivalence $\forall v.\ h(\lambda(v)) = \hat{\lambda}(f(v))$ and fwd-equivalence $\forall v.i, v'.i'.\ (v.i, v'.i') \in \text{fwd}_\lambda(v) \Leftrightarrow \widehat{v.i}, \widehat{v'.i'} \in \widehat{\text{fwd}}_{\hat{\lambda}}(\hat{v})$ □

Corollary 3. *Suppose we have well-formed $\mathcal{N}_d^s$, $\overline{\mathcal{N}_d^s}$ and $\widehat{\mathcal{N}_d^s}$, $\overline{\widehat{\mathcal{N}}_d^s}$ with effective abstractions (f, h) and $(\hat{f}, \hat{h})$, respectively. If $\overline{\mathcal{S}|\mathcal{N}_d^s}$ equals $\overline{\mathcal{S}|\widehat{\mathcal{N}}_d^s}$, for any λ, there exists label and fwd-equivalence $\hat{\lambda}$ and vice versa.*

Proof. (*i*) Suppose λ is a solution for $\mathcal{N}_d^s$. From Theorem 2, we know that solutions λ and $\overline{\lambda}$ exist that are fwd-equivalent. Additionally, $\forall v.\ h(\lambda(v)) = \overline{\lambda}(f(v))$ and $\forall v.i, v'.i'.\ (v.i, v'.i') \in \text{fwd}_\lambda(v) \Leftrightarrow (\overline{v.i}, \overline{v'.i'}) \in \overline{\text{fwd}}_{\overline{\lambda}}(\overline{v})$. From Theorem 1, we know that we have $\forall v.\ \overline{\lambda}(f(v)) = \overline{\lambda}(f_{\mathcal{S}|\mathcal{N}}(v))$ and $\forall v.i, v'.i'.\ (\overline{v.i}, \overline{v'.i'}) \in \overline{\text{fwd}}_{\overline{\lambda}}(\overline{v}) \Leftrightarrow (\overline{v.i}, \overline{v'.i'}) \in \overline{\text{fwd}}_{\mathcal{S}|\mathcal{N}\,\overline{\lambda}_{\mathcal{S}|\mathcal{N}}}(\overline{v})$. Thus, $\forall v.\ h(\lambda(v)) = \overline{\lambda}(f_{\mathcal{S}|\mathcal{N}}(v))$ and $\forall v.i, v'.i'.\ (v.i, v'.i') \in \text{fwd}_\lambda(v) \Leftrightarrow (\overline{v.i}, \overline{v'.i'}) \in \overline{\text{fwd}}_{\mathcal{S}|\mathcal{N}\,\overline{\lambda}_{\mathcal{S}|\mathcal{N}}}(\overline{v})$. Similarly, for $\widehat{\mathcal{N}}_d^s$ and $\overline{\widehat{\mathcal{N}}}_d^s$ By transitivity, we have label equivalence and fwd equivalence. □

References

[1] R. Beckett, A. Gupta, R. Mahajan, D. Walker, A general approach to network configuration verification, in: Proceedings of the Conference of the ACM Special Interest Group on Data Communication, ACM, 2017, pp. 155–168.
[2] 2016, https://www.ftmeadealliance.org/2016/02/a-day-in-the-life-of-disa/.
[3] H.H. Liu, X. Wu, W. Zhou, W. Chen, T. Wang, H. Xu, L. Zhou, Q. Ma, M. Zhang, Automatic life cycle management of network configurations, in: Proceedings of the Afternoon Workshop on Self-Driving Networks, ACM, 2018, pp. 29–35.
[4] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, T. Millstein, A general approach to network configuration analysis, USENIX NSDI, 2015.
[5] A. Gember-Jacobson, R. Viswanathan, A. Akella, R. Mahajan, Fast control plane analysis using an abstract representation, ACM SIGCOMM, 2016.
[6] S.K. Fayaz, T. Sharma, A. Fogel, V. Sekar, G. Varghese, L. Angeles, I. Osdi, S.K. Fayaz, T. Millstein, R. Mahajan, Efficient network reachability analysis using a succinct control plane representation, OSDI 2016, 2016.
[7] N.P. Lopes, P. Godefroid, K. Jayaraman, G. Varghese, Checking beliefs in dynamic networks, in: Usenix Conference on Networked Systems Design and Implementation, 2015, pp. 499–512.
[8] J.S. Jensen, T.B. Krøgh, J.S. Madsen, S. Schmid, J. Srba, M.T. Thorgersen, P-rex: Fast verification of mpls networks with multiple link failures, in: Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies, ACM, 2018, pp. 217–227.
[9] R. Beckett, A. Gupta, R. Mahajan, D. Walker, Control plane compression, in: S. Gorinsky, J. Tapolcai (Eds.), Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2018, Budapest, Hungary, August 20–25, 2018, ACM, 2018, pp. 476–489.
[10] E.M. Clarke, E.A. Emerson, A.P. Sistla, Automatic verification of finite-state concurrent systems using temporal logic specifications, ACM Transactions on Programming Languages and Systems (TOPLAS) 8 (2) (1986) 244–263.
[11] D. Awduche, J. Malcolm, J. Agogbua, M. ODell, J. McManus, Rfc2702: Requirements for traffic engineering over mpls, RFC Editor, USA, 1999.
[12] , Tiramisu: Fast and general network verification, 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20), USENIX Association, Santa Clara, CA, 2020.
[13] NUV, 2016, https://github.com/network-network/NUV.
[14] 2016, https://github.com/batfish/batfish.
[15] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P.B. Godfrey, S.T. King, Debugging the data plane with anteater, Acm Sigcomm Computer Communication Review 41 (4) (2011) 290–301.
[16] G.D. Plotkin, N. Bjørner, N.P. Lopes, A. Rybalchenko, G. Varghese, Scaling network verification using symmetry and surgery, in: ACM PLDI, 2016, pp. 69–83.
[17] P. Kazemian, G. Varghese, N. Mckeown, Header space analysis: Static checking for networks, USENIX NSDI, 2012. 9–9
[18] A. Panda, O. Lahav, K. Argyraki, M. Sagiv, S. Shenker, Verifying reachability in networks with mutable datapaths, in: USENIX NSDI, USENIX Association, Boston, MA, 2017, pp. 699–718.
[19] R. Stoenescu, M. Popovici, L. Negreanu, C. Raiciu, Symnet: Scalable symbolic execution for modern networks, in: ACM SIGCOMM, 2016, pp. 314–327.
[20] B. Tschaen, Z. Ying, T. Benson, S. Banerjee, J.M. Kang, Sfc-checker: Checking the correct forwarding behavior of service function chaining, in: 2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN), 2016.
[21] S. Spinoso, M. Virgilio, W. John, A. Manzalini, G. Marchetto, R. Sisto, Formal verification of virtual network function graphs in an sp-devops context, 9306, 2015, pp. 253–262, doi:10.1007/978-3-319-24072-5_18.
[22] A. Khurshid, X. Zou, W. Zhou, M. Caesar, P.B. Godfrey, Veriflow: Verifying network-wide invariants in real time, in: Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13), USENIX, Lombard, IL, 2013, pp. 15–27.
[23] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, S. Whyte, Real time network policy checking using header space analysis, in: Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13), USENIX, Lombard, IL, 2013, pp. 99–111.
[24] A. Horn, A. Kheradmand, M. Prasad, Delta-net: Real-time network verification using atoms, in: USENIX NSDI, USENIX Association, Boston, MA, 2017, pp. 735–749.
[25] , Plankton: Scalable network configuration verification through model checking, 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20), USENIX Association, Santa Clara, CA, 2020.
[26] R. Beckett, R. Mahajan, T. Millstein, J. Padhye, D. Walker, Don't mind the gap: Bridging network-wide objectives and device-level configurations, in: Proceedings of the 2016 ACM SIGCOMM Conference, ACM, 2016, pp. 328–341.
[27] R. Beckett, R. Mahajan, T. Millstein, J. Padhye, D. Walker, Network configuration synthesis with abstract topologies, ACM SIGPLAN Notices 52 (6) (2017) 437–451.
[28] A. El-Hassany, P. Tsankov, L. Vanbever, M. Vechev, Network-wide configuration synthesis, in: International Conference on Computer Aided Verification, Springer, 2017, pp. 261–281.
[29] K. Subramanian, L. D'Antoni, A. Akella, Synthesis of fault-tolerant distributed router configurations, Proceedings of the ACM on Measurement and Analysis of Computing Systems 2 (1) (2018) 22.

[30] A. El-Hassany, P. Tsankov, L. Vanbever, M. Vechev, Netcomplete: Practical network-wide configuration synthesis with autocompletion, 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18), USENIX Association, 2018.
[31] X. Jin, H.H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, R. Wattenhofer, Dynamic scheduling of network updates, in: ACM SIGCOMM Computer Communication Review, 44, ACM, 2014, pp. 539–550.
[32] H.H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, D. Maltz, zupdate: Updating data center networks with zero loss, in: ACM SIGCOMM Computer Communication Review, 43, ACM, 2013, pp. 411–422.
[33] B. Tian, X. Zhang, E. Zhai, H.H. Liu, Q. Ye, C. Wang, X. Wu, Z. Ji, Y. Sang, M. Zhang, D. Yu, C. Tian, H. Zheng, B.Y. Zhao, Safely and automatically updating in-network ACL configurations with intent language, in: Proceedings of the ACM Special Interest Group on Data Communication, SIGCOMM 2019, Beijing, China, August 19–23, 2019, 2019, pp. 214–226, doi:10.1145/3341302.3342088.

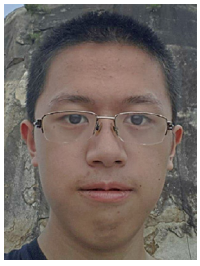
Yahui Li received her B.S. degree from the College of Software from Jilin University, China in 2015. She is now pursuing her Ph.D. at Tsinghua University. Her research concerns network verification, network testing and formal methods.

Zhiliang Wang received his B.E., M.E. and Ph.D. degrees in computer science from Tsinghua University, China in 2001, 2003 and 2006, respectively. Currently, he is an Associate Professor at the Institute for Network Sciences and Cyberspace at Tsinghua University. His research interests include formal methods and protocol testing, next-generation Internet, and network measurement.

Xia Yin received her B.E., M.E. and Ph.D. degrees in computer science from Tsinghua University in 1995, 1997 and 2000, respectively. She is a Full Professor at the Department of Computer Science and Technology at Tsinghua University. Her research interests include future Internet architectures, formal methods, protocol testing and largescale Internet routing.

Xingang Shi received his B.S. degree from Tsinghua University and his Ph.D. degree from The Chinese University of Hong Kong. He is now working at the Institute for Network Sciences and Cyberspace at Tsinghua University. His research interests include network measurement and routing protocols.

Jianping Wu received his B.S., M.S., and Ph.D. from Tsinghua University. He is a Full Professor and Director of the Network Research Center and Ph.D. Supervisor at the Department of Computer Science, Tsinghua University. Since 1994, he has been in charge of the China Education and Research Network (CERNET). He is a member of the Information Advisory Committee, Office of National Information Infrastructure, Secretariat of State Council of China and is also a Vice President of the Internet Society of China (ISC). His research interests include next-generation Internet, IPv6 deployment and technologies, and Internet protocol design and engineering.



Fangdan Ye received his B.S. degree in Computer Science and Technology from Tsinghua University, China in 2018. He is now pursuing his M.S. at Tsinghua University. His research concerns network verification and network update.

Jiangyuan Yao received his B.E. degree in engineering mechanics from Shandong University of Science and Technology, his M.E degree in network information and security from Beijing Jiaotong University and his Ph.D. degree in computer science from Tsinghua University. He is an Associate Professor at the College of Information Science and Technology at Hainan University. His research concerns formal methods, protocol testing and cyberspace security.

Han Zhang received the B.S. degree in Computer Science and Technology from JiLin University and Ph.D. in Tsinghua University. He is now working in the School of Cyber Science and Technology, Beihang University. His research concerns computer networks, network security and AI. He is a member of IEEE.